COMRADE

Ein Hochsprachen-Compiler für Adaptive Computersysteme

Von der Carl-Friedrich-Gauß-Fakultät für Mathematik und Informatik
der Technischen Universität Braunschweig

genehmigte Dissertation

zur Erlangung des Grades eines Doktor-Ingenieurs (Dr.-Ing.)

von

Dipl.-Inform. Nico Kasprzyk

Promotion am 20.06.2005

1.Referent: Prof. Dr. Ulrich Golze, Technische Universität Braunschweig
2. Referent: Prof. Dr. Christian Hochberger, Technische Universität Dresden

Eingereicht am: 01.03.2005

Vorwort

Angeregt wurde ich zum Thema dieser Dissertation während meiner Tätigkeit als wissenschaftlicher Mitarbeiter an der Abteilung Entwurf integrierter Schaltungen (E.I.S.) der Technischen Universität Braunschweig durch ein Projekt in Kooperation mit dem führenden Hersteller für Synthesewerkzeuge Synopsys und der Universität Berkeley.

Herrn Prof. Dr. U. Golze danke ich für die Betreuung dieser Arbeit, für die intensiven und fruchtbaren Diskussionen, für seine unterstützenden Hinweise sowie für die Übernahme des Referats. Herrn Prof. Dr. Christian Hochberger danke ich für die Übernahme des Koreferats und Herrn Prof. Dr. A. Koch für die wertvollen Fachgespräche über die Hardware-Synthese.

Für ihre Mitarbeit an dem Hochsprachen-Compiler für Adaptive Computersysteme COMRADE danke ich Christian Berger, Nico Feiertag, Florian Immenroth, Holger Krahn, Matthias Moll, Stefan Mücke, Michael Rock, Gerald Winter und Konrad Zufelde.

Außerdem danke ich meiner Familie, welche mir das Studium ermöglichte und mich während der Promotion unterstützend begleitete.

Kurzfassung

Rechenleistung wird klassisch durch schnellere oder zusätzliche Mikroprozessoren gesteigert. Dagegen lagert ein Adaptives Computersystem (ACS) rechenintensive Teile in rekonfigurierbare Hardware in Form eines Field-Programmable Gate-Arrays (FPGA) aus, während der Rest weiterhin als Software auf einem klassischen Mikroprozessor ausgeführt wird.

Da die Programmerstellung für ein ACS von Hand Wissen über Software- und Hardwareentwurf sowie Systemkenntnisse vom Entwickler verlangt und viel Zeit verbraucht, wurde von mir zur Erleichterung des Programmierens mit Hilfe des Compilersystems SUIF2 aus Stanford der Compiler COMRADE entwickelt. COMRADE partitioniert einen C-Quelltext in einen Software-Teil und eine Hardware-Konfiguration für ein ACS. Dabei werden automatisch Regionen für eine Hardware-Beschleunigung bestimmt und Hardware-Konfiguration und Software unter besonderer Berücksichtigung einer gemeinsamen Kommunikation zu einem lauffähigen Ganzen zusammengesetzt.

COMRADE kann durch die Nutzung einer Modulbibliothek für Hardware-Komponenten und Konfigurationsdateien leicht an neue Zielarchitekturen angepasst werden. Er unterstützt den vollen C-Sprachumfang. Eine verbesserte Hardware-Software-Partitionierung vergrößert den synthetisierten Hardware-Anteil. Weiterhin sind in COMRADE Heuristiken implementiert, welche die Nachteile der hohen Rekonfigurationszeit von aktuellen Logikbausteinen durch Zusammenfassen mehrerer Hardware-Teile zu Konfigurationen umgehen. Dadurch kann ein Großteil der während der Programmausführung auf einem ACS sonst nötigen Konfigurationen eingespart werden.

Für die Steuerung der Hardware wurden Konzepte von bestehenden Datenflussmaschinen weiterentwickelt. Neben Hardware-Pipelines wird die spekulative Berechnung von Ergebnissen unterstützt. Aus mehreren Ergebnissen wird das richtige für die weitere Berechnung ausgewählt. Durch ein neuartiges auf Petri-Netzen aufbauendes Ausführungsmodell ergibt sich gegenüber herkömmlichen spekulativen Mechanismen ein Laufzeitgewinn. Hardware-Operatoren mit fester und variabler Laufzeit werden unterstützt.

Die Anwendung verschiedener Optimierungen auf Hochsprachenebene zeigte wesentlichen Einfluss auf die Größe der erzeugten Hardware. So konnte nachgewiesen werden, dass einerseits der Einfluss jeder einzelnen Optimierung Vorteile erbringt, die Kombination von Optimierungen aber den Effekt noch erhöht. Durch die in COMRADE implementierten Optimierungen konnten der Hardware-Verbrauch der Operatoren wie auch die Anzahl der Speicherzugriffe wesentlich reduziert werden. Ressourcenverbrauch und Geschwindigkeit der Hardware werden so positiv beeinflusst.

Inhalt

Kapitel 1

Einleitung

Wer ärgert sich nicht, wenn er eine Anwendung auf einem modernen Rechner nicht in ausreichender Geschwindigkeit ausführen kann oder bei der Arbeit von Lüftern gestört wird, welche die in den Rechnern erzeugte Wärme abtransportieren müssen? Auch wenn die Rechner immer schneller werden: der Trend zu mehr Wärmeentwicklung pro Chipfläche bei Prozessoren kann auch in Zukunft bei sich verkleinernden Transistoren anhalten (Bild 1.1), wenn die Technik der Chipherstellung sich nicht ändert. Um dem Anwender aber ein Gerät anbieten zu können, welches ausreichend schnell, leise und dennoch preiswert ist, müssen die Entwickler immer zwischen zwei Extremen unterscheiden.

Zum einen haben sie die Möglichkeit, für die Geräte Standard-Prozessoren zu verwenden. Diese sind zwar relativ preiswert und Anwendungen lassen sich durch die Verfügbarkeit von Werkzeugen schnell entwickeln, doch können sie nur wenige Befehle pro Takt ausführen und benötigen für diese relativ viel Energie. Weil der Anwender schnelle Programme wünscht, müssen die Prozessoren mit hohen Taktraten laufen und produzieren Wärme, die derzeit durch teilweise große Lüfter an die Umgebung abgegeben werden muss.

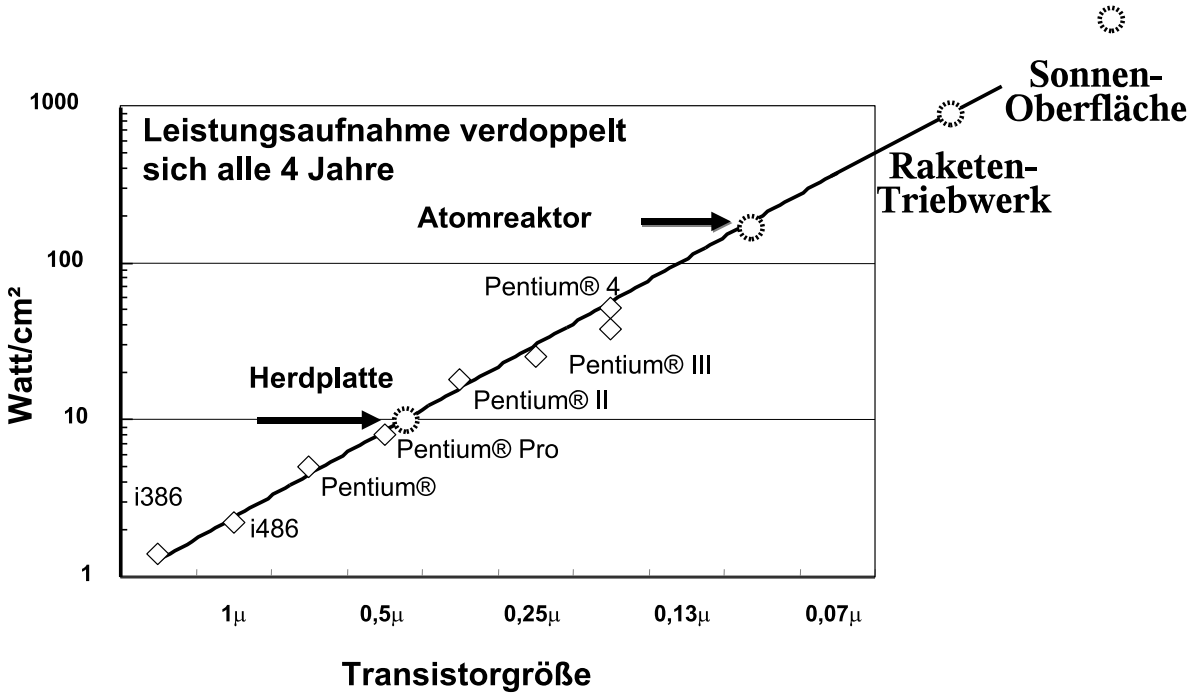


Bild 1.1 Exponentiell steigende Wärmeabgabe von Prozessoren [Poll99]

Auf der anderen Seite können sich die Entwickler aber auch für einen Chip entscheiden, der genau auf die Anwendung zugeschnitten ist. Auf diesem Logikbaustein können verhältnismäßig viele Befehle zur gleichen Zeit ausgeführt werden. Jeder der Befehle benötigt wenig Energie und erzeugt somit wenig Wärme. Durch neue Logikbausteine hat man sogar die Möglichkeit, die Funktion auch noch nach der Herstellung zu ändern. Somit können die Vorteile auch für verschiedene Anwendungen ausgenutzt werden. Leider ist aber die Aufwand zum Erstellen einer Anwendung für einen Logikbaustein immer noch dramatisch höher als der für einen Prozessor.

Eine Lösung, welche dem Anwender große Rechenressourcen zur Verfügung stellt und außerdem auch dem Entwickler mehr Freiheit in der Zusammensetzung von Rechnern gibt, ist die Kombination von Prozessor und Logikbaustein zu so genannten Adaptiven Computersystemen. In ihnen können die Vorteile beider Bausteine genutzt werden: die gute Anpassungsfähigkeit von Prozessoren für viele Anwendungen mit wenig Leistungsbedarf und die hohe Rechenleistung von Logikbausteinen für wenige Anwendungen mit hohem Leistungsbedarf.

Für solche Systeme ist durch die Verfügbarkeit von mehr Bausteinen leider auch ein erhöhter Entwicklungsaufwand für Anwendungen gegenüber bisher verfügbaren Rechnern notwendig. Der Entwurf der Schaltungen für den Logikbaustein sowie die Einbeziehung der Synchronisation der Arbeit beider Bausteine kann bei der Programmerstellung von Hand sehr lange dauern und ist somit zu teuer. Eine Lösung dieses Problems liegt in der Verwendung von Compilern, welche wie bisher eine Programmbeschreibung in einer Programmiersprache in ein ausführbares Programm übersetzen. Dabei müssen durch diesen Compiler Aufgaben wie die Unterteilung des Anwendungsprogramms auf beide Bausteine sowie die Erstellung einer optimierten Schaltung für den Logikbaustein übernommen werden.

Diese Arbeit beschreibt detailliert die im Forschungscompiler COMRADE[1] integrierten Algorithmen zur Handhabung der Probleme, welche bei der Übersetzung von C-Quelltext in ein auf einem Adaptiven Computersystem lauffähiges Programm auftreten.

1.1 Aufbau der Arbeit

In Kapitel zwei der Arbeit wird der allgemeine Aufbau von Adaptiven Computersystemen beschrieben. Hierbei wird die Verwendung dieser für Berechnungen mit hohem Leistungsanspruch motiviert, und es werden verschiedene Typen solcher Systeme vorgestellt. Im Weiteren werden unterschiedliche Arten von rekonfigurierbaren Bausteinen beschrieben, welche diesen Systemen zugrunde liegen. Den Abschluss des Kapitels bilden die Beschreibungen unterschiedlicher Entwurfsmethoden von Programmen für Adaptive Computersysteme inklusive des in dieser Arbeit beschriebenen Hochsprachencompilers COMRADE. In den sich anschließenden Kapiteln werden für die Compilierung von C wichtige Schritte genauer erklärt.

Das dritte Kapitel beschäftigt sich mit der Aufteilung eines gegebenen C-Quelltexts in Hardware- und Software. Dabei wird ein mehrstufiges Verfahren eingeführt, welches auf-

[1] COMpiler füR ADaptive systemE

grund von Laufzeitinformationen aus Teilen des Quelltexts Kandidaten für eine Hardware-Ausführung bestimmt und nach deren Konvertierung in einen Datenpfad entscheidet, ob ein jeweiliger Kandidat letztendlich wirklich in Hardware realisiert werden soll.

Im vierten Kapitel stellt diese Arbeit die Umwandlung der Hardware-Kandidaten in Datenpfade vor. In einem Verfahren aus zwei Schritten wird aus den Kandidaten zuerst ein Datenflussgraph erzeugt und in einem zweiten Schritt für diesen ein Kontroller erzeugt. Dieser Kontroller entscheidet bei der Ausführung des aus Datenflussgraph und Kontroller bestehenden Datenpfads, wann welche Operation im Datenflussgraph ausgeführt wird. Mit Hilfe des Kontrollers ist nicht nur eine Ausführung als Pipeline möglich. Es können darüber hinaus Teile spekulativ berechnet und bei Nichtbenutzung spekulativ begonnener Berechnungen diese auch wieder deaktiviert werden. Die in der Datenpfad-Generierung eingebundenen Grundlagen und Mechanismen sind wichtiger Bestandteil des Kapitels.

Das Thema des fünften Kapitels ist die Auswirkung von Optimierungen auf einer C-Quelltext ähnlichen Ebene der Programmdarstellung auf die generierten Datenpfade. Dazu werden zu Beginn die für COMRADE zur Verfügung stehenden Analysen des Quelltexts vorgestellt. Darauf aufbauend werden Optimierungen beschrieben, deren Auswirkungen auf die Größe der Datenpfade mit experimentellen Ergebnissen belegt werden.

Die im sechsten Kapitel eingeführten Heuristiken haben das Ziel, durch geschicktes Zusammenfassen mehrerer Datenpfade zu einer Konfiguration Zeit für die benötigten Konfigurationen während der Programmausführung zu sparen und damit die Programmausführung selbst zu verkürzen. Hierzu werden zuerst die für die Heuristiken wichtigen Daten und Hilfsgraphen beschrieben. Daran schließt sich die Vorstellung unterschiedlicher als sinnvoll erachteter Heuristiken an, deren Verwendbarkeit am Ende des Kapitels mittels Testdaten belegt wird.

Das siebte Kapitel fasst die in dieser Arbeit beschriebenen Themen zusammen und gibt einen Ausblick auf mögliche zukünftige Entwicklungen von COMRADE. Im Anhang befinden sich in Erweiterung des Kapitels vier für die Generierung des Kontrollers ergänzende Informationen.

1.2 Hinweise für den Leser

Einige der folgenden Kapitel wurden durch einen Abschnitt *Erweiterungsmöglichkeiten* mit in Zukunft denkbaren Ergänzungen der zurzeit bestehenden Programmerzeugung für ein ACS durch COMRADE ergänzt. Alle dabei direkt mit der Implementierung in COMRADE zusammenhängenden Vorschläge werden mit der Beschreibung der Implementierung [Kasp05] durch eine Verweisnummer (CE_xx[2]) in Zusammenhang gebracht. Dieser Nummer sind in der Beschreibung mehrere Positionen im Quellcode mit den Erklärungen zu erforderlichen Änderungen zugeordnet. Dies erleichtert die Verbesserung der Implementation bei der Fortführung des Projekts.

[2] COMRADE Erweiterung

Die Referenzen zur Arbeit wurden in zwei Klassen eingeteilt: gedruckte Erzeugnisse und Internetreferenzen. Die Referenzen zu Internetressourcen wurden unterstrichen und sind in einem getrennten Abschnitt im Literaturverzeichnis aufgeführt. Da sich die Quellen im Internet schnell ändern und auch verloren gehen können, wurden diese auch auf der beiliegenden CD bereitgestellt.

Kapitel 2

Adaptive Computersysteme

In vielen computerbasierten Anwendungsbereichen wachsen die Anforderungen an Rechenleistung in größerem Maße als dies durch die auf dem Markt verfügbaren Systeme befriedigt werden kann. Dem soll der Aufbau von Systemen bestehend aus Prozessoren und rekonfigurierbaren Bausteinen Abhilfe schaffen. In diesem Abschnitt wird die Verwendung von Adaptiven Computersystemen motiviert und deren Aufbau vorgestellt.

Für viele Anwendungen sind Hardware-Implementierungen die besten Lösungen [DaPR01]. Vor allem bei Programmen, in denen viele Operationen in einem Zeitschritt parallel ausführbar sind, laufen Hardware-Implementierungen um ein Vielfaches schneller als auf herkömmlichen Prozessoren. Dieser Geschwindigkeitsgewinn kommt durch die unterschiedliche Bearbeitung von Programmen auf beiden Einheiten. Klassische Prozessoren bearbeiten Programme meist seriell. Auf einem Logikbaustein sind demgegenüber sehr viele Operationen parallel ausführbar.

Bei einem Rekonfigurierbaren Logikbaustein (RL) hat man darüber hinaus noch die Möglichkeit, das rechnende System auf sich ändernde Anforderungen anzupassen. Eine Vielzahl von Anwendungen hat die Eignung dieser Systeme für einen breiten Einsatz bereits bewiesen [LiEA00].

Aber warum werden nicht alle Prozessoren durch einen RL ersetzt? Der Grund hierfür ist im zurzeit hohen Aufwand zur Programmerstellung und Programmierung dieser Hardware zu sehen. Anders als die Software-Entwicklung für einen klassischen Prozessor erfordert die Programmentwicklung für einen RL bis zum heutigen Zeitpunkt noch tief greifende Systemkenntnisse. Es gibt zwar Ansätze, Betriebssysteme für einen RL zu verwenden, doch ist die Entwicklung noch nicht so weit vorangeschritten wie bei klassischen Prozessorsystemen [NoEA03].

Des Weiteren lohnt sich der Programmentwurfs-Aufwand vor allem nicht für kontrollflussdominierte Teile eines Programms. Ohne **spekulative Ausführung** in RL-Konfigurationen werden aufgrund des Fehlens von Parallelität keine Geschwindigkeitsgewinne gegenüber einem modernen Prozessorsystem erzielt.

Adaptive Rechner nutzen die Vorteile von Bausteinen aus beiden Gruppen. Zum einen werden Prozessoren eingesetzt, welche einen festgelegten Befehlssatz besitzen. Das zweite Element solcher Strukturen sind rekonfigurierbare Bausteine, welche ihr Verhalten während der Ausführung eines Programms ständig an die aktuellen Bedürfnisse des Systems anpassen.

2.1 Definitionen

Zum Verständnis des aktuellen Kapitels sollen einige Begriffe erläutert werden. Diese sind hauptsächlich aus [Golz04] übernommen worden. Die **Rekonfigurierbarkeit** von Logikbausteinen bezeichnet die Fähigkeit, eine logische Struktur aus verbundenen Recheneinheiten ohne Chip-Fertigung oder physikalischen Hardware-Umbau nur durch Programmierung zu realisieren. Arten von RL und deren Eignung für Adaptive Computersysteme werden ausführlich in Abschnitt 2.2 behandelt. Findet die Rekonfiguration während der Ausführung eines Programms statt, dann spricht man von **dynamischer Rekonfiguration**.

Besteht die Möglichkeit der teilweisen Umprogrammierung eines RL im laufenden Betrieb, so bezeichnet man dies als dynamische, **partielle Rekonfiguration**.

2.1.1 Anforderungen an Adaptive Computersysteme

Damit ein Adaptives Computersystem zur schnellen Berechnung von Programmen eingesetzt werden kann, sollte es idealerweise verschiedene Eigenschaften erfüllen. Diese Eigenschaften erweitern sinnvoll alle an klassische Computersysteme gestellte Forderungen, um eine maximale Performance zur Verfügung stellen zu können. Da ein ACS im Allgemeinen als ein klassisches Computersystem betrachtet werden kann, welches durch einen RL erweitert wurde, betreffen die Anforderungen an ein ACS speziell den RL und die damit verbundenen Probleme.

Die erste Forderung an ein ACS ist die *Programmierbarkeit des RL im System*. Für ein ACS ist es wichtig, dass der RL im laufenden Betrieb des ACS von einer Systemkomponente selbst konfiguriert werden kann. Kein zusätzlich an das ACS angeschlossenes Gerät soll zur Programmierung nötig sein. Für den gewinnbringenden Einsatz des RL im System sollte die Programmierzeit in einem vorteilhaften Verhältnis zu der Laufzeit von Programmteilen auf dem RL liegen.

Die zweite Forderung an ein ACS ist die *optimale Einbindung des RL in das System*. Es soll hierfür ein schneller Zugriff auf möglichst viele Systemressourcen des Systems für den RL ermöglicht werden. Wichtigste Ressource ist hierbei der Speicher. Außerdem kann es sinnvoll sein, weitere Komponenten wie Systembusse und Interrupts zugänglich zu machen. Damit wären beispielsweise Ausgaben auf einem Bildschirm aus dem RL möglich. Im derzeitigen Entwicklungsstadium ist die optimale Anbindung an den flüchtigen Systemspeicher im ACS noch eines der Hauptziele.

Die dritte Forderung an ein ACS ist die *effiziente Steuerung der Programmausführung auf der RL*. Dabei sollte der Systemdesigner Mechanismen zur Verfügung stellen, die einen schnellen Austausch von Daten und Ereignissen zwischen dem RL und anderen rechnenden Einheiten gestattet.

Ein ideales ACS würde diesen Forderungen zufolge dem RL alle Möglichkeiten eröffnen, die in klassischen Systemen ein Prozessor besitzt (Speicherzugriff, Zugriff auf Interrupts, Hochgeschwindigkeitskommunikation zwischen Prozessoren). Außerdem wäre der RL schnell und teilweise konfigurierbar. Die Verbindung zwischen dem RL und anderen rechnenden

Einheiten sollte zumindest so schnell sein wie in klassischen Mehrprozessorsystemen zwischen den Prozessoren.

2.1.2 Realisierungen von Adaptiven Computersystemen

Adaptive Computersysteme können auf unterschiedliche Weise realisiert werden. Die aktuellen Forschungen beschäftigen sich weitestgehend mit dem Zugriff des RLs auf die Systemressource flüchtiger Speicher. Deswegen sollen hier ACS-Varianten aufgrund dieser Eigenschaft unterschieden werden.

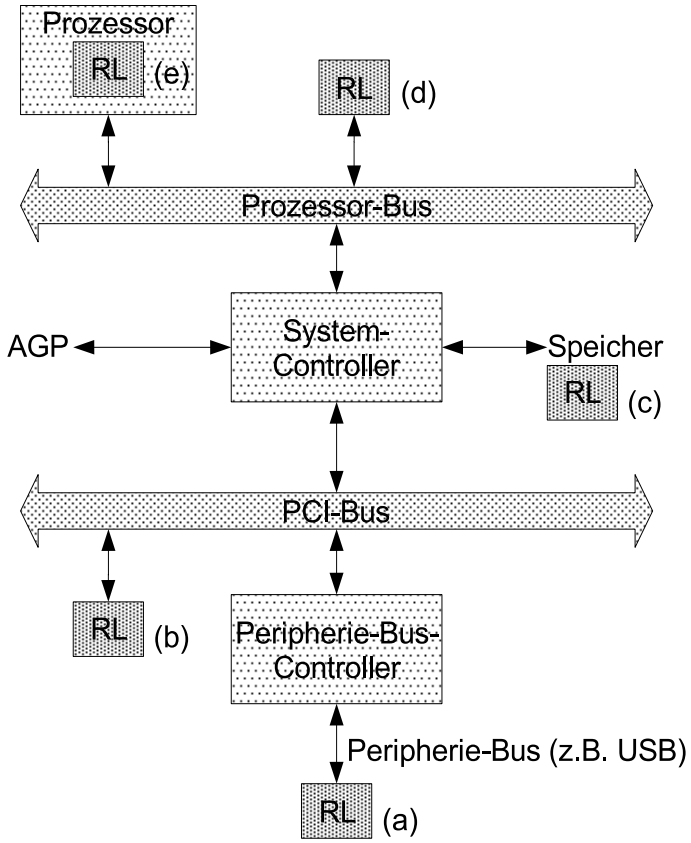


Bild 2.1 Einbindung eines RL in ein System, (a) Freistehender RL, (b) Angeschlossene RL, (c) RL statt Speicher, (d) RL als Coprozessor, (e) RL im Prozessor integriert

Einerseits kann der RL über einen verhältnismäßig langsamen externen Ausgang eines Rechners an den Prozessor eines Computersystems angeschlossen sein (Freistehende RL, Bild 2.1a)[LGDR03]. Da sich aber die Kommunikation zwischen Prozessor und RL als beschränkender Faktor herausgestellt hat, werden Realisierungen mit schnelleren Kommunikationsmöglichkeiten bevorzugt.

So gibt es die Möglichkeit des Anschlusses des RL über einen internen Peripheriebus (angeschlossene RL, Bild 2.1b). Als ein besonderes Beispiel kann hierbei die in [LeEA01] beschriebene Architektur angesehen werden, bei welcher ein Speicherbaustein eines handelsüblichen PC durch eine mit einem **Field-Programmable Gate-Array** (FPGA) bestückte

Leiterplatte ersetzt wird (Bild 2.1c). Besonderer Vorteil dieser Architektur ist die schnelle Kommunikation zwischen Prozessor und FPGA. Es müssen aber äußere Programmiergeräte verwendet werden. Außerdem kann das FPGA nicht direkt auf Speicher zugreifen.

Bei weiterer Annäherung zwischen RL und Prozessor greifen beide Bausteine über getrennte Caches gleichberechtigt auf den Systembus und damit noch schneller auf den Speicher zu, wobei besonders auf die Konsistenz der Daten in den Caches geachtet werden muss (Bild 2.1d). Dieser Aufbau ist ähnlich einem Coprozessorsystem. Probleme kann es hierbei nur noch durch die verteilten Caches geben. Abhilfe bringt die Verwendung von Prozessoren, welche in Mehrprozessorsystemen eingesetzt werden. Diese stellen geeignete Mechanismen zur Verfügung, welche die Konsistenz der Caches gewährleisten. Leider muss auch auf dem RL Logik vorhanden sein, welche die dazu notwendigen Protokolle implementiert. Dadurch fehlt kostbare Fläche für Berechnungen.

Bei einem optimalen ACS sind RL, Prozessor und Cache sowie verschiedene Steuereinheiten wie z.B. für die Interrupt-Behandlung auf einem Chip integriert (Bild 2.1e). Dadurch entsteht eine enge Kopplung zwischen den Komponenten.

Als Grundlage für den in dieser Arbeit vorgestellten Compiler COMRADE (Abschnitt 2.3.3) gehen wir von einem ACS aus, welches jede der zuvor beschriebenen Architekturen haben kann.

2.1.3 Das Adaptive Computersystem ACE2

Als Grundlage unserer ACS-Forschungen dient die Prozessorkarte ACE2 [Koch00]. Diese wird mit einem Erweiterungs-Board (ADM-XRC) betrieben, welches mit einem Xilinx XCV1000 FPGA bestückt und über den PCI-Bus mit der Prozessorkarte verbunden ist (Bild 2.2). Dieses System hat also eine angeschlossene RL. Das FPGA hat über den PCI-Bus vollen Zugriff auf den Systemspeicher und kann über ihn somit parallel zum Prozessor verfügen. Zur Vereinfachung der Kommunikation zwischen Prozessor und FPGA sind auf den Boards Konverter zwischen den Bus-Protokollen PCI und i960 eingebaut (PLX-9080) eingebaut, da das i960-Protokoll einfacher in einem FPGA implementierbar ist als das PCI-Protokoll. Die beiden XC4085-FPGAs der ACE2-Karte werden aufgrund zu geringer Ressourcen nicht verwendet.

Auch wenn sich dieses System noch weit vom Idealbild eines ACS auf einem Chip befindet, so sind hiermit schon grundlegende Versuche zur ACS-Programmierung durchführbar. Besonders vorteilhaft ist die Anpassung des Betriebssystems RTEMS auf diese Plattform. Diese ermöglicht einen transparenten Zugriff auf die Systemressourcen und stellt ein API zum Zugriff auf den rekonfigurierbaren Teil der Plattform bereit.

Als Nachteil der ACE2-Plattform findet der Speicherzugriff von Prozessor und FPGA nicht über ein gemeinsames Interface statt. Inkonsistenzen zwischen den Daten im Cache des Prozessors und dem Speicher können dazu führen, dass das FPGA mit falschen Daten arbeitet. Dieser Fall muss bei der Programmerstellung für die ACE2-Karte berücksichtigt werden.

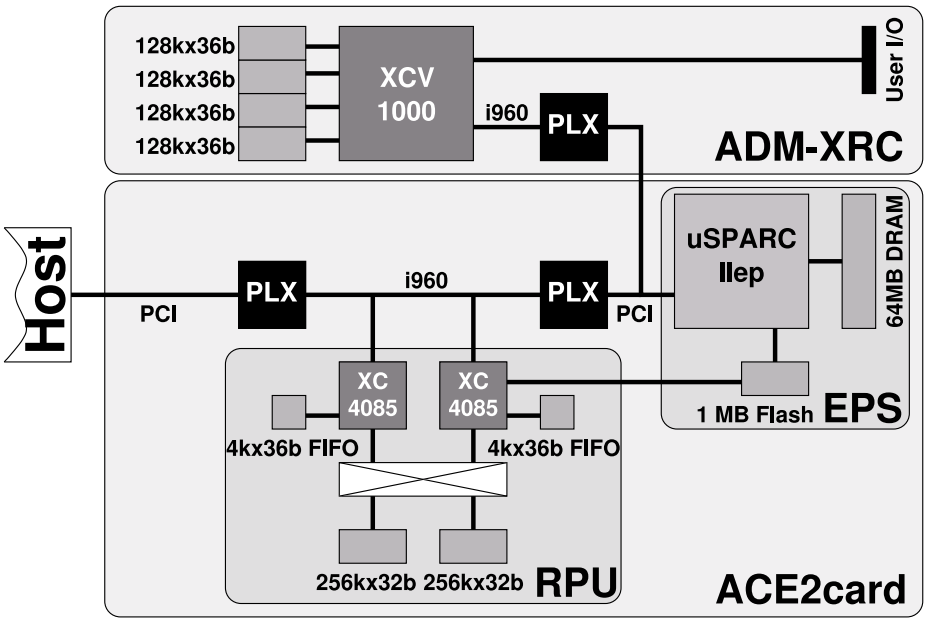


Bild 2.2 ACE2-Architektur mit Embedded Processor Subsystem (EPS), Reconfigurable Processing Unit (RPU) und Erweiterungs-Board ADM-XRC

2.2 Rekonfigurierbare Logikbausteine

Der wichtigste Bestandteil eines ACS sind konfigurierbare Logikbausteine. Diese teilen sich noch in die Gruppen der einmal und mehrmals konfigurierbaren ein. Einmal konfigurierbare Logikbausteine sind für die Anwendung in adaptiven Rechnern ungeeignet, da sie nicht während der Programmlaufzeit angepasst werden können.

In den folgenden Abschnitten soll die Eignung von derzeit gefertigten sowie in der Entwicklung befindlichen RLs beurteilt werden. Ein Faktor hierbei ist die Bitbreite der auf dem RL realisierbaren Operatoren. Prototypen für die auf einem ACS zu realisierenden Programme sind normalerweise Programmiersprachen wie FORTRAN oder C. Die Bitbreite der Datentypen in diesen Programmiersprachen ist größtenteils acht Bit oder ein Vielfaches davon. Operatoren mit nur einem Bit Breite treten dagegen selten auf. Somit werden im RL größtenteils Operationen implementiert, die auf mindestens acht Bit breiten Bussen arbeiten.

Weiterhin sollte ein RL partiell und schnell rekonfigurierbar sein. Damit kann sichergestellt werden, dass der Wechsel zwischen verschiedenen Konfigurationen die Programmausführung nicht zu stark verlangsamt.

2.2.1 Field-Programmable Gate-Arrays

Zurzeit sind von den kommerziell erhältlichen Logikbausteinen vor allem FPGAs dazu geeignet, für ein ACS als rekonfigurierbare Bausteine angewendet zu werden. Der Aufbau eines FPGA besteht hauptsächlich aus einer regulären Struktur von CLBs, welche durch ein Netz

von programmierbaren Verbindungen miteinander verbunden sind. Die CLBs bestehen meist aus programmierbaren Lookup-Tables sowie Registern. Auch die Verdrahtungen zwischen den CLBs kann programmiert werden. Diese Architektur lässt eine feingranulare Programmierung zu und macht somit FPGAs universell anwendbar für die Implementierung von digitalen Schaltungen. Im Gegensatz zu CPLDs können so unterschiedliche Operationen durch die auf FPGAs verfügbaren rekonfigurierbaren Logikblöcke (CLBs) ohne verhältnismäßig großen Flächenverlust realisiert werden.

Durch die Feinkörnigkeit das FPGA-Konfiguration wiederum müssen bei der Anwendung für ein ACS für jeden Operator verhältnismäßig viele CLBs gleichartig programmiert werden. Dies liegt an der schon erwähnten Bitbreite in den Anwendungsprogrammen für die ACS-Programmierung. Diese Vervielfachung von Informationen vergrößert den Programmieraufwand und damit auch die Programmierzeit [FeKT01].

Der Nachteil der langsamen Rekonfigurierbarkeit wird in einigen FPGA-Architekturen durch eine partielle Rekonfiguration teilweise behoben. Beschränkungen bestehen aber weiterhin durch eine zu grobkörnige partielle Rekonfigurierbarkeit.

FPGAs sind aufgrund der beschriebenen Eigenschaften nur bedingt in einem ACS einsetzbar.

2.2.2 Spezialisierte rekonfigurierbare Logikbausteine

Die Nachteile von FPGAs für die Implementierung einer Vielzahl von Algorithmen haben in den letzten Jahren zur Entwicklung von einigen neuen Architekturen geführt. Ihre Anwendbarkeit für ein ACS soll hier diskutiert werden. Die Architekturen stehen bisher nun als Intellectual-Property-Blöcke (IP-Blöcke) zu Verfügung und könnten somit in ein ACS-on-Chip eingefügt werden.

Die Firma Elixent [Elix01] versucht, mit ihren Systemen multimediale Signalverarbeitung zu vereinfachen. Vor allem der Aufwand zur Herstellung von ASICs für Multimedia-Systeme in kleinen Serien soll durch eine verstärkte Einführung rekonfigurierbarer Techniken reduziert werden.

In ihren Reconfigurable Algorithm Processors (RAP) werden bisher in DSPs implementierte Designs durch rekonfigurierbare Datenpfade ersetzt. Dadurch wird die Wiederverwendbarkeit von Chip-Designs erhöht sowie der Entwicklungszyklus für ein neues Produkt verkürzt.

Ein RAP besteht schematisch betrachtet hauptsächlich aus einem Array von 4-Bit ALUs und Register/Buffer-Blöcken, die zu unterschiedlichen Datenpfad-Breiten zusammengefasst werden können. Daten können mit den acht umgebenden Einheiten ausgetauscht werden. Die ALUs sind dazu durch Switchboxes miteinander verbunden, welche als Kreuzpunkt oder als Rekonfigurationsspeicher verwendet werden können. Weiterhin sind Speicherblöcke im Design platzierbar (Bild 2.3).

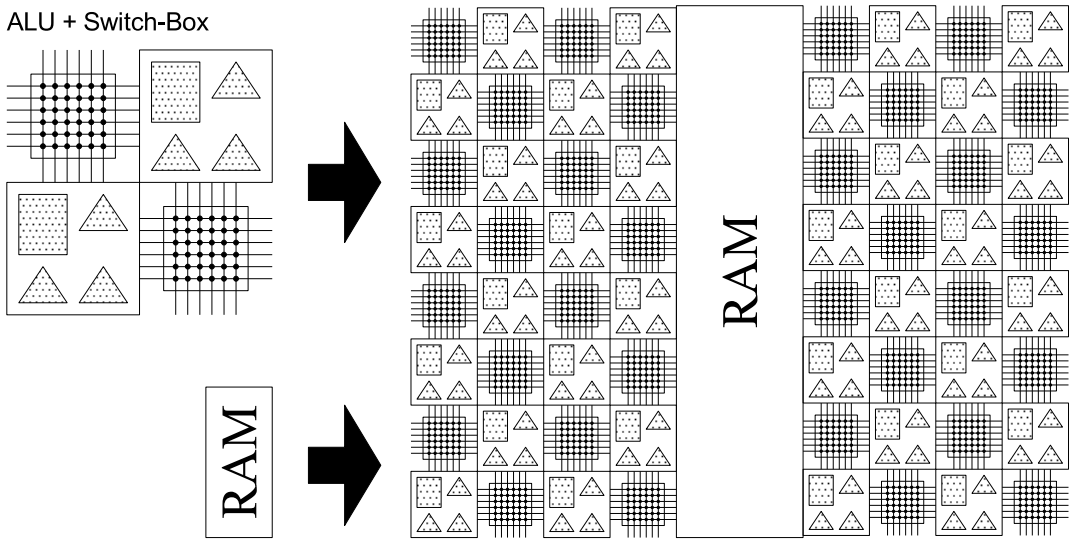


Bild 2.3 RAP: Prozessor-Array

Der Vorteil dieser Architektur liegt in den Operatorbreiten und der Beschränkung auf nur wenige Operationen in der ALU gegenüber den CLBs auf einem FPGA. Damit benötigt man weniger Rekonfigurationsbits und folglich auch weniger Rekonfigurationszeit.

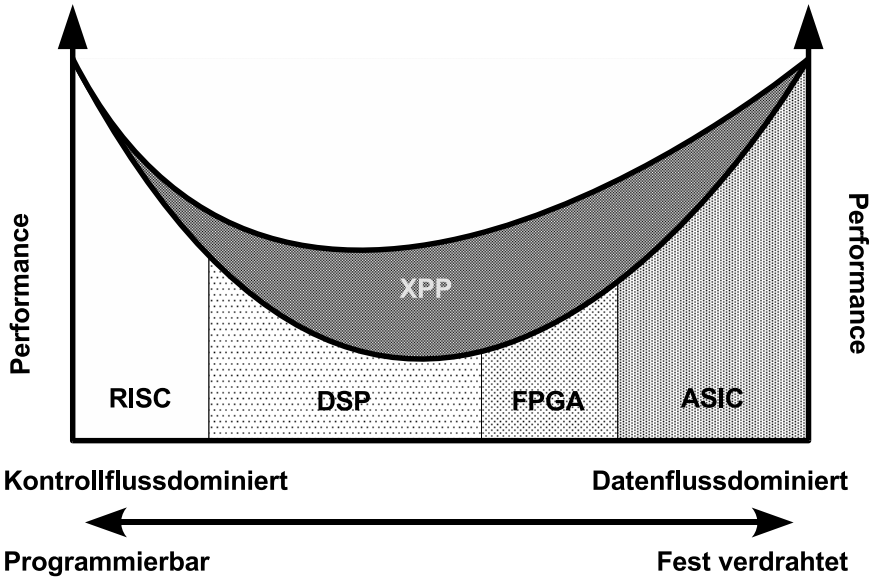


Bild 2.4 Avisiertes Marktsegment der XPP-Architektur

Eine andere Architektur ist die eXtreme Processing Platform (XPP) der Firma PACT. Diese soll teilweise Nachteile bisheriger Technologien reduzieren, um damit in deren Marktsegmente einzusteigen (Bild 2.4). Dies soll vor allem durch eine Änderung der Sichtweise auf die Programmausführung erreicht werden [BeVo03].

Statt der Ausführung von einzelnen Maschinenbefehlen auf einzelnen Daten sollen in der XPP-Architektur Datenströme durch Elemente berechnet werden, die sich ständig rekonfigurieren. Jede Aufgabe wird in Teilaufgaben zerlegt, wobei jede einzelne Aufgabe durch die auf dem Chip verfügbaren Ressourcen gelöst werden kann. Nach Abschluss einer Teilaufgabe wird eine neue Konfiguration auf den Chip geladen und der Ergebnisdatenstrom der letzten Teilaufgabe bearbeitet. Dieses Wechseln von Rekonfiguration und Bearbeitung wird solange wiederholt, bis die Aufgabe gelöst wurde.

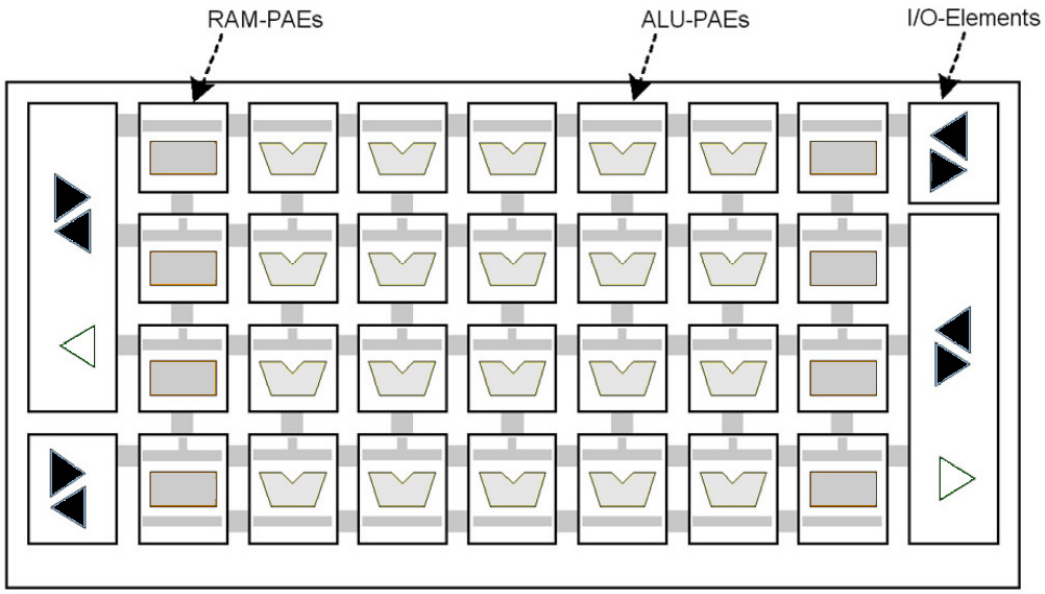


Bild 2.5 Schematischer Aufbau der XPP-Architektur

Der schematische Aufbau der XPP-Architektur ist einem FPGA sehr ähnlich. Statt der CLBs werden im XPP Processing Array Units (PAEs) benutzt (Bild 2.5). Die Befehlssätze der PAEs sind ähnlich einem DSP strukturiert. Busse verbinden die PAEs nur in horizontaler Richtung. Somit können in horizontaler Richtung Daten und Signalen zwischen den PAEs und auf dem Chip befindlichen RAMs ausgetauscht werden. Vertikal können zwischen den Bussen Daten und Signale nur mit Hilfe der PAE- und RAM-Zellen ausgetauscht werden.

Prinzipiell ist die XPP-Architektur sehr gut für die Anwendung in einem ACS geeignet. Vor allem die schnelle Rekonfiguration und der Einsatz von Rekonfigurationsmanagern verspricht eine gute Einbindung in ein ACS. Nachteilig ist die Kommunikationsstruktur auf dieser Architektur. Vertikale Daten- und Kontrollflüsse sind nur durch die Verwendung von PAE-Ressourcen erstellbar. Da es aber bei der Implementation von Schaltungen auch vertikale Verbindungen geben muss, wird wahrscheinlich ein Großteil der PAEs nur als Daten- oder Signalweg verwendet.

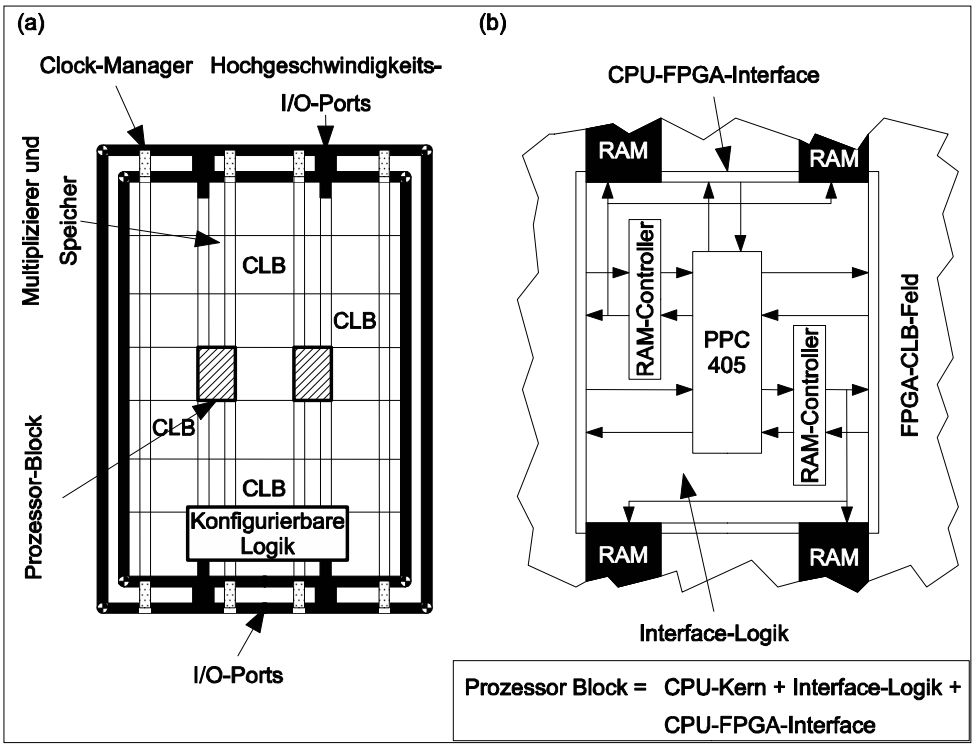


Bild 2.6 (a) Blockdiagramm des Virtex-II Pro Chips, (b) Prozessoranbindung ([Xili03])

2.2.3 Programmierbarer Logikbaustein und Mikroprozessor auf einem Chip

Einige Hersteller für rekonfigurierbare Chips sind dazu übergegangen, Mikroprozessorkerne zu integrieren.

Die Firma Xilinx bietet mit den Virtex-II Pro und Virtex-II Pro X [Xili03] FPGAs mit integrierten PowerPCs auf einem Chip an. Durch die Verbindung von Prozessor und RL auf einem Chip werden gute Voraussetzungen für ein ACS gegeben. Das Blockdiagramm (Bild 2.6a) zeigt die Anordnung der Prozessorblöcke im Zentrum des CLB-Arrays. Außerdem enthält der Chip noch RAM sowie 18x18-Integer-Multiplizierer. Die Prozessoren selbst können auf Speicher auf dem Chip zugreifen, die Clock des Chips steuern sowie die CLBs programmieren (Bild 2.6b).

Für den Zugriff auf Ressourcen außerhalb des Chips wie Speicher gibt es keine speziellen Bausteine auf dem Chip. Alle diese Zugriffe müssen durch IP-Blöcke im RL ausgeführt werden. Damit gehen kostbare Ressourcen für die Nutzung in einem ACS verloren.

Einen Schritt weiter geht Garp[Haus97]. Der Aufbau dieser Architektur (Bild 2.7) lässt erkennen, dass sie direkt für ein ACS entwickelt wurde. Auf den externen Datenspeicher wird von beiden rechnenden Komponenten des Chips aus über einen gemeinsamen Cache zugegriffen. Dateninkonsistenzen beim Speicherzugriff sind somit ausgeschlossen. Des Weiteren besteht das rekonfigurierbare Array aus einer FPGA-ähnlichen Struktur, wobei die Operatorbreiten zwei Bit sind. Zur Rekonfiguration wird direkt auf den Speicher zugegriffen. Eine

Implementation dieses Chips wäre hervorragend für den Einsatz in einem ACS geeignet. Leider steht bisher noch kein lauffähiger Chip zur Verfügung.

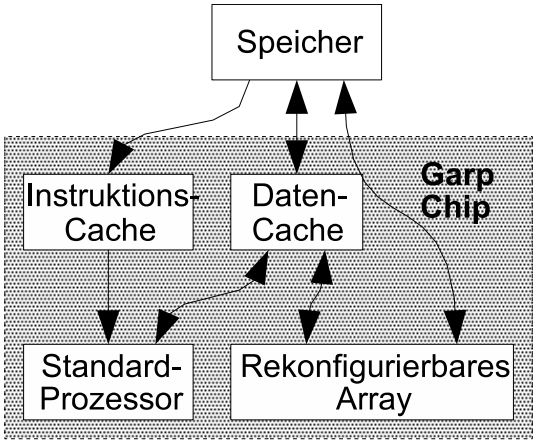


Bild 2.7 Blockdiagramm des Garp-Chips (aus [Haus97])

2.3 Entwurfswerkzeuge für Adaptive Computersysteme

Das Hauptziel der Programmentwicklung für ein ACS liegt in der Ausnutzung der auf dem RL verfügbaren Parallelität der Programmausführung und der damit verbundenen Beschleunigung der Programmausführung. Außerdem können Programmteile für die Ausführung auf dem RL durch Einbinden von Konstanten oder das Wissen über Datentypen spezialisiert werden. Dies kann zu einer Beschleunigung der Ausführung führen. Deshalb sollten vor allem Teile eines Programmquellcodes mit einem hohen Beschleunigungspotential durch den RL auf dieser ausgeführt werden.

Bei allen Ansätzen für die Entwicklung von Programmen für ein ACS wird davon ausgegangen, dass schon eine Beschreibung des auf dem ACS ausgeführten Programms in einer Hochsprache (also rein algorithmisch) vorliegt. Je nach Ansatz wird der Quellcode manuell oder automatisch übersetzt.

2.3.1 Hardware-Software-Codesign

Beim HW-SW-Codesign wird der in einer Hochsprache vorliegende Quellcode von einem Entwickler nach Kriterien wie Ressourcenverbrauch im RL und Beschleunigungspotential durch den RL untersucht. Er sucht dann Regionen aus dem Quellcode aus, welche sich besonders für die Realisierung in HW lohnen. In diesem Schritt wird der Entwickler meist durch verschiedene Werkzeuge unterstützt [HeEr98, ViEA02]. Aufgrund der erlangten Informationen werden nun Regionen für die HW-Realisierung bestimmt.

Diese HW-Regionen werden dann in eine Beschreibung konvertiert, welche für den Ziel-RL geeignet sind (Hardware-Beschreibungssprachen wie Verilog). Dabei wird oft manuell die

funktionale Beschreibung in eine Register-Transfer-Logik (RTL) umgesetzt, um eine effektivere HW-Realisierung zu erhalten. Die Schnittstellen zwischen dem HW- und dem SW-Teil des Programms werden per Hand direkt auf das ACS angepasst.

Die Programmierung eines ACS durch HW-SW-Codesign stellt an den Entwickler derzeit noch sehr hohe Anforderungen, da er neben Kenntnissen aus dem Bereich der SW-Herstellung auch Erfahrungen mit dem HW- und dem Systementwurf haben muss. Deswegen versuchen verschiedene Ansätze, den Entwickler auf verschiedenen Stufen der Entwicklung zu unterstützen.

Eine Erleichterung des Entwicklungsflusses sind Beschreibungssprachen, welche gleichsam für HW wie auch für SW geeignet sind [Erns97, LeeA03]. So kann man manuell, halbautomatisch oder automatisch den Quellcode partitionieren und die jeweils für HW und SW vorgesehenen Teile automatisch implementieren lassen. Das Gesamtsystem lässt sich simulieren und somit erhält man verhältnismäßig schnell Resultate über die Güte einer Partitionierung und kann diese bei Bedarf ändern.

2.3.2 Automatische Programmentwicklung aus Hochsprachen

Um die Programmentwicklung auch Entwicklern ohne Kenntnisse des ACS selbst zugänglich zu machen, wird versucht, die Programmbeschreibung in der Hochsprache automatisch in ein auf dem ACS lauffähiges Programm zu übersetzen. Der Compiler muss dazu selbständig entscheiden, welche Programmregionen in HW ausgeführt werden sowie welche Schnittstelle für die Kommunikation zwischen HW und SW verwendet wird. Des Weiteren werden Ansätze benötigt, welche die funktionale Beschreibung in einer Hochsprache automatisch und effizient in eine HW-Realisierung übersetzen können.

Meist wird als Beschreibungssprache für die automatische Programmentwicklung die Programmiersprache C gewählt, da diese einen verhältnismäßig kleinen Sprachumfang gegenüber Java oder C++ enthält und außerdem viele Problembeschreibungen in C vorliegen. Neben der aus dem HW-SW-Codesign bekannten automatischen Partitionierung der Programmbeschreibung [YeSB00, DiEA01] wird auch die Strategie der Duplikation von Programmregionen zur HW-Beschleunigung genutzt [CaWa98, LiEA00]. Hierbei werden für HW-Realisierungen geeignete Regionen dupliziert und stehen dann als SW- und HW-Realisierungen während der Programmausführung zur Verfügung. Dadurch kann auf die SW-Version zurückgegriffen werden, wenn während der Programmausführung beispielsweise aufgrund von Datenabhängigkeiten oder Problemen mit dem RL keine HW-Realisierung ausgeführt werden kann.

Außerdem muss für ein ACS auch das Problem der Synthese von HW aus einer Hochsprache gelöst werden. Eine Lösung ist die Einschränkung [ViEA02] oder Erweiterung [WeLu01] der Programmiersprache zur Verbesserung der Synthetisierfähigkeit. Die automatische Synthese für Konstrukte der Hochsprache ist meist kompliziert und erzeugt schlechtere Designs als die manuelle.

2.3.3 Compilerfluss von COMRADE

Zum Test der in dieser Arbeit beschriebenen Methoden wurde der Compiler COMRADE (Bild 2.8) implementiert. Er setzt auf Teilen des vorhandenen Entwurfsflusses des Projekts Nimble [LiEA00] auf. Ziel ist die automatische Erzeugung von effizienten Programmen für ein ACS aus einer Programmbeschreibung in C. Er benutzt die Duplikation von Programmteilen als Grundlage für alle verwendeten Algorithmen.

Ein Großteil der Algorithmen wird auf dem Kontrollflussgraph (CFG) des zu bearbeitenden Quellcodes ausgeführt. Der Kontrollflussgraph entsteht aus der Konvertierung des aus dem C-Quellcode erstellten sowie optimierten Abstrakten Syntaxbaums. Im CFG werden Regionen des Programms dupliziert, welche sich für eine HW-Implementierung eignen (Abschnitt 3.2.1).

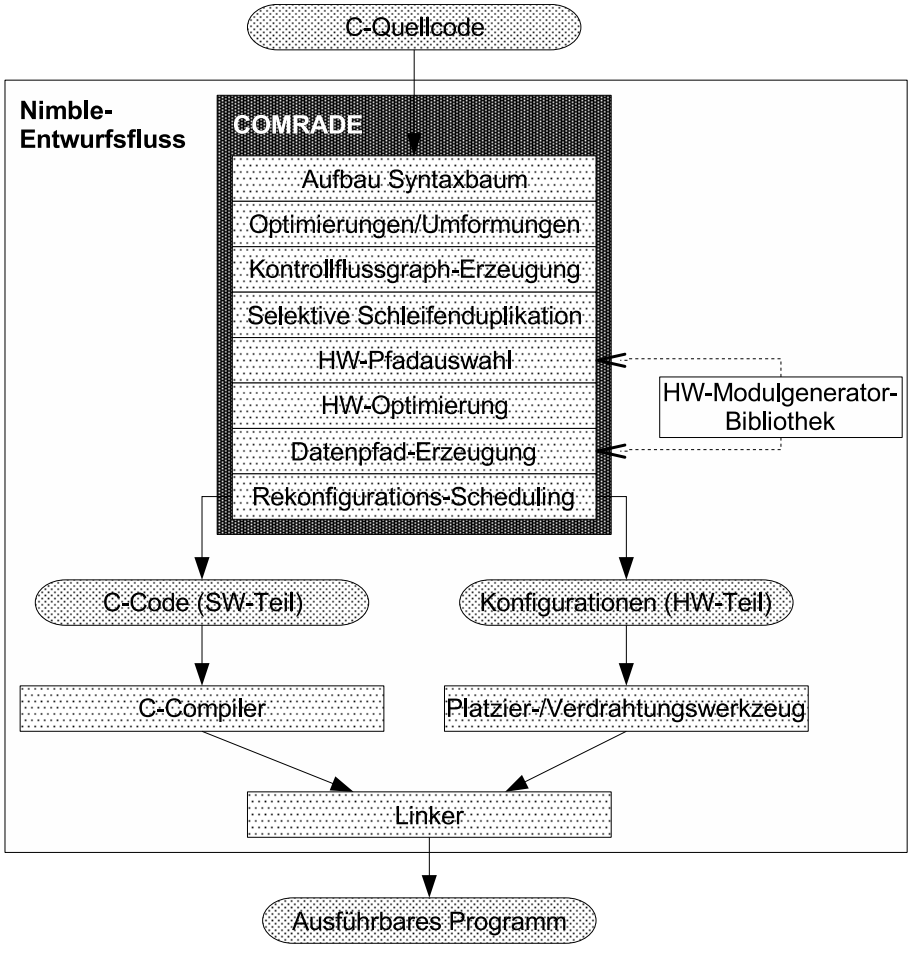


Bild 2.8 Einbettung von COMRADE in den Nimble Entwurfsfluss

Aufgrund von Informationen über die HW-Realisierung von einzelnen Operatoren in den duplizierten Regionen sowie Laufzeitinformationen werden nun alle Teile entfernt, welche sich nicht für eine HW-Ausführung eignen. Die entfernten Teile müssen im endgültigen Pro-

gramm durch den Prozessor ausgeführt werden. Alle in den Regionen verbliebenen Operationen werden in Datenpfade übersetzt (Kapitel 4). Erst nach der Erzeugung der Datenpfade wird entschieden, ob die HW-Realisierung aufgrund von Beschleunigungspotential und Ressourcenverbrauch auf dem RL wirklich benutzt werden kann. Erst zu diesem Zeitpunkt sind alle notwendigen Informationen vorhanden. Die entstehenden Datenpfade sind oft so klein, dass sie die Ressourcen des RL nicht vollständig ausnutzen. Deshalb wird im letzten Compilerschritt versucht, mehrere Datenpfade zu einer Konfiguration zusammenzufassen. Am Ende der Bearbeitung durch COMRADE wird der SW-Teil daraufhin wieder als C-Code herausgeschrieben. Die Konfigurationen werden Werkzeugen übergeben, die aus ihnen Bibliotheken erzeugen. Diese Bibliotheken werden mit dem aus dem SW-Teil entstehenden Programm zusammengebunden und bilden das auf dem ACS lauffähige Programm.

Kapitel 3

Aufteilung in Hardware und Software

Dieses Kapitel beschreibt die verwendete Strategie zur Aufteilung eines Anwendungsprogramms in Regionen für die HW- und SW-Ausführung. Ziel dieser Aufteilung ist die Beschleunigung des Anwenderprogramms durch das Beschleunigungspotential des RL in einem ACS. Kriterien für diese Auswahl sind die Ausführungswahrscheinlichkeit von Regionen im Anwendungsprogramm, der Beschleunigungsfaktor durch eine HW-Ausführung sowie Kommunikationszeiten zwischen dem Prozessor und der RL.

Die folgenden Abschnitte stellen die für die Aufteilung in HW- und SW-Ausführung notwendigen Analyseverfahren vor und zeigen die in COMRADE verwendete Strategie zur Auswahl der HW-Regionen. Die Anwendbarkeit der vorgestellten Strategie wird am Ende des Kapitels durch Messungen an Benchmark-Programmen belegt.

3.1 Grundlegende Analysen

Für die Auswahl von HW-Regionen werden in COMRADE für jedes ACS spezifische sowie für das Anwendungsprogramm charakteristische Daten zugrunde gelegt. Einige dieser Daten sind nicht oder nur mit einem verhältnismäßig hohem Aufwand automatisch zu bestimmen. So sollte die Messung der Kommunikationszeiten zwischen Prozessor und RL sowie der Rekonfigurationszeit des RLs nicht durch den Compiler vorgenommen werden. Außerdem müsste der Compiler dazu auch Zugang zu einem ACS haben. Dies ist bisher noch nicht vorgesehen.

Die **Kommunikationszeit zwischen Prozessor und RL t_K** ist die Zeit, welche zur Übertragung eines Datenworts vom Prozessor über einen Kommunikationskanal an den RL oder in der Gegenrichtung benötigt wird. Da diese nicht konstant ist, wird die mittlere Kommunikationszeit verwendet.

Die **Rekonfigurationszeit des RLs t_R** ist die Zeit, welche zwischen der Initialisierung einer Rekonfiguration durch den Aufruf einer Bibliotheksfunktion auf dem ACS bis zur Rückmeldung der beendeten Rekonfiguration an das laufende Programm vergeht.

Ausgangspunkt für die Aufteilung in HW und SW ist ein in der Programmiersprache C geschriebenes Anwendungsprogramm. Die folgenden grundlegenden Definitionen sind an [Acka00] angelehnt und für das weitere Verständnis notwendig.

Definition 1: Sei B eine Menge von **Befehlen** und $l \in \mathbb{N}$. p: $\{1,..., l\} \rightarrow B$ ist eine **Prozedur** (der **Länge** l) und für $1 \leq n \leq l$ ist n die n-te **Anweisung** der Prozedur.

In dieser Arbeit werden die Anweisungen der Programmiersprache C betrachtet. Dies sind Zuweisungen und bedingte Sprünge. Außerdem werden die Schlüsselworte mit den in Klammern stehenden Ausdrücken der Kontrollflusskonstrukte `for`, `while`, `do-while`, `if` und `switch` als Anweisungen betrachtet. So ist beispielsweise `if( a < 24 )` eine `if`-Anweisung und `if( a < 24 ) { i++; }` das `if`-Konstrukt.

Da diese Konstrukte den Kontrollfluss eines Programms beeinflussen, soll damit der Kontrollfluss definiert werden. Grundlegend wichtig hierfür ist die Definition für das Ende der Kontrollflusskonstrukte `for`, `while`, `if` und `switch`. Diese als **Konstruktende** bezeichnete Anweisung wird an die Stelle der Prozedur platziert, an der das Kontrollflusskonstrukt endet (schließende Klammer). Aus einem `do-while`-Konstrukt entstehen für die Schlüsselworte `do` und `while` jeweils getrennte Anweisungen.

Definition 2: Die Anweisung n ist eine **Kontrollflussverzweigung**, falls $p(n)$ eine `for`-, `while`-, `do`-, `if`- oder `switch`-Anweisung ist.

Definition 3: Die Anweisung n ist eine **Kontrollflussvereinigung**, wenn eine der folgenden Bedingungen zutrifft:

i. $p(n)$ ist ein Konstruktende oder
ii. $p(n)$ ist die `while`-Anweisung eines `do-while`-Konstrukts oder
iii. n ist das Ziel von mindestens einem unbedingten Sprung (`goto`).

In der Programmiersprache C sind keine Befehle für bedingte Sprünge enthalten. Die Vereinigung von Kontrollflüssen am Ende von `if`-, `while`- und `for`-Anweisungen wird durch das Konstruktende behandelt.

Definition 4: Gegeben sei eine Prozedur der Länge $l \in \mathbb{N}$. (b, e) mit $b \in \mathbb{N}$ und $e \in \mathbb{N}$ ist ein **Basisblock** der Prozedur, wenn gilt:

i. $1 \leq b \leq e \leq l$,
ii. alle i mit $b+1 \leq i \leq e-1$ sind weder eine Kontrollflussvereinigung noch eine Kontrollflussverzweigung,
iii. b ist der Eintrittpunkt in eine Prozedur oder eine Kontrollflussvereinigung sowie
iv. e ist die letzte Anweisung einer Prozedur oder eine Kontrollflussverzweigung.

Ein Basisblock ist somit eine maximale Sequenz von Anweisungen einer Prozedur, in welcher sich der Kontrollfluss nicht aufspaltet oder zusammenfließt. Die erste Anweisung des Basisblocks ist der Beginn der Prozedur oder das Ziel einer Kontrollflussvereinigung. Die letzte Anweisung des Basisblocks ist entweder das Ende der Prozedur oder eine Kontrollflussverzweigung.

Für die Arbeit eines Compilers sind neben Basisblöcken auch Blöcke von Bedeutung, welche keine maximale Sequenz von Anweisungen oder keine Anweisung enthalten.

Definition 5: Sei p eine Prozedur der Länge $l \in \mathbb{N}$ und $Anw=\{1, ..., l\}$ die Menge aller Anweisungen von p. Eine disjunkte Überdeckung von Anw definiert eine **Menge von Kontrollflussblöcken** $Kb=\{k1, ..., kn\}$, wenn gilt:

i. Für alle **Kontrollflussblöcke** $k \in Kb$ gibt es b_k und e_k mit $1 \leq b_k \leq e_k \leq l$ oder $k=\varnothing$.

ii. Ist ein Kontrollflussblock nicht leer und i eine Anweisung mit $b+1 \leq i \leq e-1$, so ist sie weder eine Kontrollflussvereinigung noch eine Kontrollflussverzweigung.

Auf dieser Grundlage definiert sich der Kontrollflussgraph:

Definition 6: Sei p eine Prozedur. $Kg = (Kb,E)$ ist ein **Kontrollflussgraph** (**von** p) und $a \rightarrow b \in E$ eine **Kontrollflusskante** (in Kg), wenn gilt:

i. Kg ist ein gerichteter und zusammenhängender Graph,
ii. Kb ist eine Menge von Kontrollflussblöcken von p sowie
iii. für jeden von der Kontrollflussverzweigung x abgehenden Zweig nach y und jeden von einem Vorgänger x zu einer Kontrollflussvereinigung y führenden Zweig gibt es eine Kontrollflusskante $x \rightarrow y \in E$.

Die Kontrollflusskanten werden nur betreten, wenn die mit ihnen verbundenen Bedingungen erfüllt sind. Ein **Nachfolger**-(Kontrollflussblock) s eines Kontrollflussblocks b ist derjenige, für den die Kante $b \rightarrow s$ existiert. Ein **Vorgänger**-(Kontrollflussblock) p eines Kontrollflussblocks b ist derjenige, für den die Kante $p \rightarrow b$ existiert. Ein Kontrollflussblock eines CFG ohne Vorgänger ist ein **Eingangsblock** des CFG. Ein Kontrollflussblock eines CFG ohne Nachfolger ist ein **Ausgangsblock** des CFG. Wir setzen in COMRADE voraus, dass bearbeitete CFGs genau einen Eingangsblock und genau einen Ausgangsblock enthalten. Prozeduren mit mehr als einem Eingangs- und/oder Ausgangsblock können leicht durch Hinzufügen von leeren Kontrollflussblöcken in einen solchen CFG umgeformt werden.

Definition 7: $Rg=(Kb_{Rg},E_{Rg})$ ist eine **Region** des Kontrollflussgraphen $Kg=(Kb,E)$, wenn gilt:

i. $Kb_{Rg} \subseteq Kb$ und $E_{Rg} \subseteq E$.
ii. $E \cap Kb_{Rg}^{\ 2} \subseteq E_{Rg}$.
iii. Rg ist ein zusammenhängender Graph.

Eine Region ist ein zusammenhängender Ausschnitt aus einem CFG, zu dem alle Kanten zwischen allen enthaltenen Kontrollflussblöcke gehören. Die Kontrollflussblöcke einer Region können summiert mehr als einen Vorgänger und mehr als einen Nachfolger haben, die nicht zur Region aber zum CFG gehören.

Des Weiteren unterscheiden sich die Kontrollflusskanten im Hinblick auf ihre Flussrichtung (zum Eingangsblock oder Ausgangsblock eines CFG). Zur Bestimmung dieser Unterschiede sind folgende Definitionen wichtig:

Definition 8: Gegeben sei ein CFG (Kb,E). Eine Knotenfolge $(kb_1,...,kb_n)$ mit $kb_i \rightarrow kb_{i+1} \in E$, für die $kb_1,..., kb_{n-1}$ paarweise verschieden sind, heißt **Pfad von** kb_1 **nach** kb_n der **Länge** n. Ein Pfad bildet einen **Zyklus**, wenn kb_1 Nachfolger von kb_n ist.

Definition 9: Ein Kontrollflussblock a **dominiert** einen Kontrollflussblock b, wenn alle Pfade vom Eingangsblock zu b durch a führen. Der Kontrollflussblock a wird als **Dominator von** b bezeichnet. Ein Kontrollflussblock $a \neq b$ **dominiert** b **strikt**, wenn a den Kontrollflussblock b dominiert. Ein Kontrollflussblock a ist ein **direkter Dominator** von b, wenn a den Kontrollflussblock b strikt dominiert und alle strikten Dominatoren von b auch a dominieren.

Mit Hilfe dieser Definitionen können die Flussrichtungen und mit ihnen Schleifen in CFGs eindeutig bestimmt werden.

Definition 10: Dominiert für eine Kante $a \rightarrow b \in E$ eines CFG der Kontrollflussblock b den Kontrollflussblock a, dann ist sie eine **zurückgerichtete Kante**.

Zurückgerichtete Kanten sind dem Kontrollfluss vom Eingangsblock zum Ausgangsblock entgegengerichtet. Sie sind wichtig für die Erkennung von Schleifen. Schleifen als häufig ausgeführte Regionen eines CFG sind das Ziel von Hardwarebeschleunigungen in COMRADE.

Definition 11: Gegeben sei die zurückgerichtete Kante $n \rightarrow h \in E$ im CFG Kg. Die **natürliche Schleife** dieser Kante mit **Schleifenkopf** h ist die Region eines CFG von Kg,

i. welcher aus den Kontrollflussblöcken $\{h, kb_0, ..., kb_m, n\}$ besteht und
ii. in dem es Pfade von allen Kontrollflussblöcken kb_i mit i=0,..,m nach n ohne den Kontrollflussblock h gibt.

```
int main() {
  int i;
  int S=42;
  for( i=0; i<=10; i=i+1 ) {
    if( i%2 == 0 ) {
      S=S+i;
    } else {
      printf( "%i", i );
    }
  }
  printf( "Ergebnis S=%i\n", S );
}
```

Beispiel 3.1 for-Schleife

Die Begriffe sollen an einem Beispiel erläutert werden (Beispiel 3.1). Dieses addiert alle geraden Zahlen zwischen Null und Zehn auf eine Ganzzahl-Variable. Alle ungeraden Zahlen werden während der Bearbeitung ausgegeben. Der aus diesem C-Programm entstehende CFG ist in Bild 3.2 zu sehen.

In der Darstellung wurden zum besseren Verständnis des CFG die Initialisierung und die Schrittweite der `for`-Anweisung virtuell als zwei kursiv gedruckte Anweisungen eingefügt. Der Kontrollflussblock kb_1 enthält somit 3 Anweisungen, wobei `i=0` hier nur virtuell vorhanden ist und aus der `for`-Anweisung stammt. Die Kante $kb_6 \rightarrow kb_2$ ist eine zurückgerichtete Kante, da kb_2 den Kontrollflussblock kb_6 dominiert. Die Kontrollflussblöcke kb_2, kb_3, kb_4, kb_5 und kb_6 bilden eine natürliche Schleife (grau unterlegt), da alle Pfade von diesen Kontrollflussblöcken zu kb_6 nicht kb_2 enthalten. Der Kontrollflussblock kb_3 wird in diesem Beispiel durch die Kontrollflussblöcke kb_1, kb_2 und kb_3 dominiert, wobei kb_1 und kb_2 strikt dominieren und kb_2 der direkte Dominator ist.

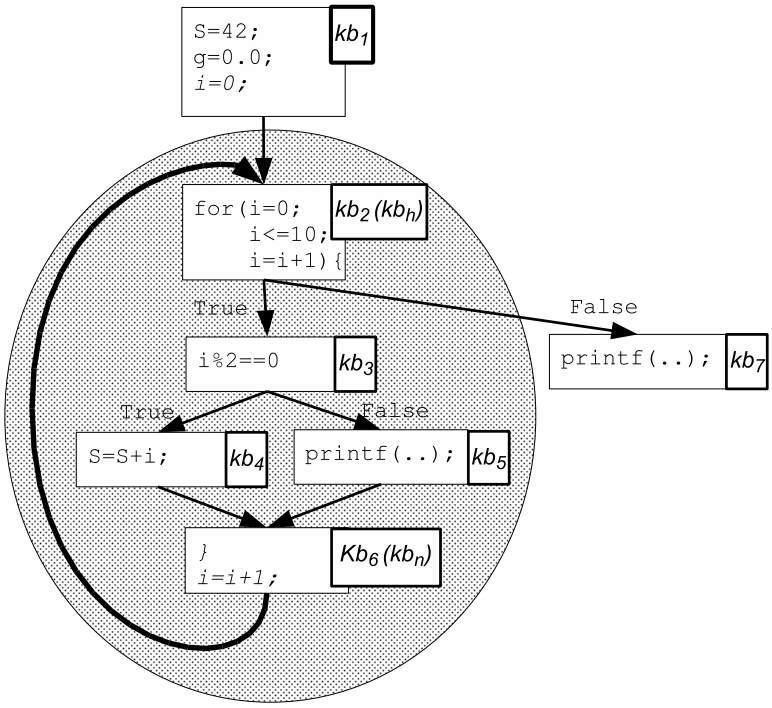


Bild 3.2 CFG für Beispiel 3.1

3.1.1 Profiling

Durch das **Profiling** eines Programms wird die Ausführungsanzahl für jede einzelne Anweisung des Programms bestimmt. Meist wird zu diesem Zweck ein Anwendungsprogramm mit Zählanweisungen instrumentiert, compiliert und danach mit brauchbaren Eingabedaten ausgeführt (**dynamisches Profiling**). Während der Programmausführung werden Informationen gesammelt, welche durch Programme wie gprof [GrKM82] den Anweisungen im Anwendungsprogramm Ausführungsanzahlen zuweisen kann. Es können aber auch Simulatoren verwendet werden, um ein Anwendungsprogramm ohne Compilation auszuführen.

Weiterhin ist es möglich, diese Daten ohne die Ausführung eines Programms zu ermitteln (**statisches Profiling**). Dies kann z.B. wichtig sein, wenn ein Anwendungsprogramm nicht terminiert oder mit Daten arbeiten muss, welche die Ausführungsanzahl stark beeinflussen. Weiterhin muss man die Anwendungsprogramme nicht compilieren, ausführen und nochmals compilieren. Vor allem in eingebetteten Systemen ist der Erhalt von Laufzeitinformationen sehr schwierig.

Hier kann man nun auf Heuristiken zurückgreifen, welche ohne Ausführung eines Programms Laufzeitinformationen abschätzen. Dabei werden für die von Verzweigungen in einem CFG ausgehenden Kanten Ausführungswahrscheinlichkeiten festgelegt. Diese beruhen auf Erfahrungswerten [BaLa93, HMBG93, MAGH94, WuLa94, DeCH98] oder auf der Analyse der Wertebereiche von Variablen [Patt95].

Der größte Vorteil des statischen Profilings gegenüber dem dynamischen ist die Geschwindigkeit. Außerdem kann es für jede Architektur ausgeführt werden und ist vollständig unabhängig von Eingabedaten. Nachteilig ist die weniger gute Bestimmung von Ausführungs-

anzahlen der Anweisungen in einem Programm. Es wurde aber gezeigt, dass die Voraussage bei einer Reihe von Benchmark-Programmen sehr gut mit den dynamisch ermittelten und gewichteten Werten übereinstimmen [WuLa94]. Für die auf Profiling basierenden Optimierungen haben die abgeschätzten Werte nur geringe Auswirkungen [HMBG93].

Die durch das Profiling entstehenden Daten können vielfältig für Optimierungen in SW-Compilern benutzt werden [BaZo93, BrCo94, CMCH92, ChMH91]. Auch für die Auswahl und die Optimierung von HW sind die durch das Profiling ermittelten Daten wichtig. Der Aspekt der Optimierung soll in diesem Kapitel nicht beleuchtet werden. Deshalb werden die im Profiling ermittelten Daten zu Ausführungsbeiwerten umgeformt.

Definition 12: Sei anz_n die Ausführungsanzahl der Anweisung $p(n)$ und $amax=max\{anz_1,...,anz_l\}$ die maximale Ausführungsanzahl im Programm der Länge l. Der **Ausführungsbeiwert** ab_n der Anweisung $p(n)$ ist definiert durch $ab_n=anz_n/amax$.

Der Grund der Einführung des Ausführungsbeiwerts liegt in der Normierung der in jedem Anwendungsprogramm unterschiedlichen Ausführungsanzahlen auf das Intervall [0,1]. Somit können die Laufzeitinformationen besser in ein Gewicht eingearbeitet werden.

3.1.2 Ressourcenverbrauch auf dem rekonfigurierbaren Logikbaustein

Ein weiteres wichtiges Entscheidungskriterium für die Auswahl von HW-Regionen sind die durch die Operatoren im Anwendungsprogramm verwendeten Ressourcen und deren Laufzeit auf der Ziel-RL. Die Bestimmung dieser findet im CFG statt. Behilflich ist die Darstellung jeder Anweisung im CFG durch einen **Abstrakten Syntaxbaum** (AST, Bild 3.3b)[Much97]. Alle in diesem AST vorkommenden Ausdrücke, die zur Anweisung $p(i)$ gehören und die Nummer j in diesem AST haben, werden mit $ausd_{i,j}$ bezeichnet.

In diesem AST müssen nun alle Operatoren auf ihren Ressourcenverbrauch und ihre Laufzeit (kurz HW-Informationen) auf dem Ziel-RL untersucht werden. Dazu werden alle Kontrollflussblöcke des CFG besucht und für alle relevanten Operatoren in den ASTs der Kontrollflussblöcke die Ressourcen abgefragt und für jeden Operator gespeichert. In dem in Bild 3.3b dargestellten AST sind beispielsweise nur die Operatoren + und `cv` (hier: Konvertierung einer Ganzzahlvariablen in eine Gleitkommavariablen) von Bedeutung. Das Laden von Variablenwerten (`Ld`) und die Zuweisung an eine Variable (=) werden in der HW für den Ziel-RL in Verdrahtung umgesetzt.

Die Anfragen zu den HW-Informationen der Operatoren werden in COMRADE an den Modulgenerator GLACE [KoKa2002] gerichtet. Diese Anfragen werden vereinfacht durch die formalisierte Schnittstelle FLAME [Koch99], welche Anfragen auf verschiedenen Beschreibungsebenen erlaubt. COMRADE erhält die HW-Informationen durch Anfragen auf der Synthese-Ebene. Als Ergebnis einer Anfrage erhält COMRADE ein *HW-Modul*, wenn für den Ziel-RL ein solches für den Operator generiert werden kann. Aus dem AST werden dazu für jeden Operator folgende Fragen beantwortet:

- Welche Operation wird durch den Operator ausgeführt? Dazu werden die im Anwendungsprogramm vorhandenen Operatoren (+, –, ...) auf die in FLAME formalisierte Operatorbeschreibung (`add`, `sub`, ...) abgebildet.
- Welche Eingänge und Ausgänge hat ein Operator? Aufgrund der Typinformationen aus dem AST können die geforderten Port-Breiten und Port-Typen für die HW-Module bestimmt werden. Dabei müssen gegebenenfalls Informationen aus dem C-Anwendungsprogramm abgeändert werden. So hat der Shift-Operator in C zwei 32-Bit-Argumente. Das Modul hat aber einen 32-Bit-Eingang für den zu schiebenden Wert und einen 5-Bit-Eingang für die Steuerung.

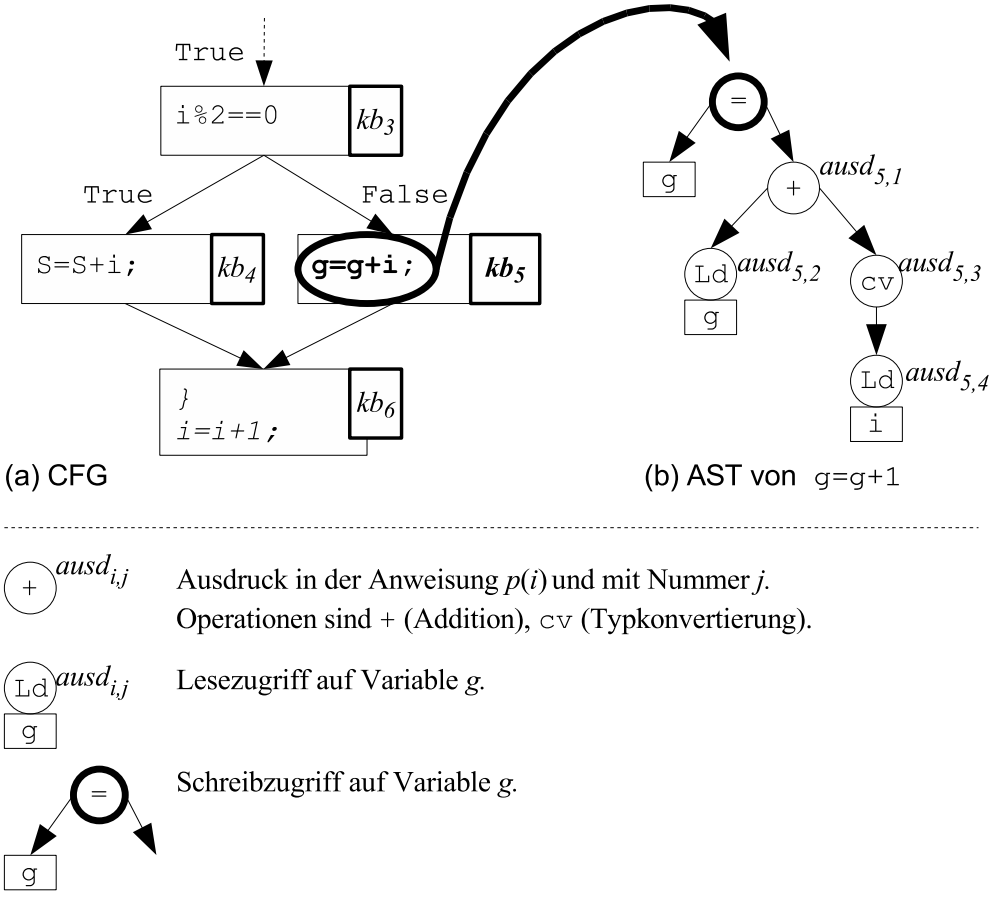


Bild 3.3 Abstrakter Syntaxbaum einer Anweisung im CFG

Die Zuordnung von Modulen zu Operationen wird folgendermaßen beschrieben:

Definition 13: Sei *Md* eine Menge von HW-Modulen (inkl. dem **Nicht-Modul**) und *Ausd* die Menge der Ausdrücke eines Programms. Ein **FLAME-Zugriff** *fl*: *Ausd*→*Md* ordnet jedem Ausdruck des Programms ein HW-Modul zu.

Für jedes HW-Modul stehen die verbrauchten Ressourcen auf dem Ziel-RL sowie die Laufzeit des Moduls zur Verfügung. Ist das Ergebnis des FLAME-Zugriffs das Nicht-Modul, dann steht kein Modul für einen Ausdruck zur Verfügung.

Zur Beschleunigung der Anfragen an den Modulgenerator wird ein Cache benutzt, da durch die Gleichheit vieler Operationen in C ein Großteil der verhältnismäßig langsamen Anfragen an den Modulgenerator eingespart werden können [KoKa02].

3.2 Hardware-Auswahl

Die HW-Auswahl gliedert sich in drei getrennte Schritte, die in den folgenden Abschnitten detailliert beschrieben werden. Im ersten Schritt, der *Schleifenduplikation*, werden alle Schleifen mit Beschleunigungspotential in der HW dupliziert.

Zwischen diesem und dem folgenden Schritt der HW-Auswahl wird die Markierung aller Operatoren mit HW-Informationen in den duplizierten Regionen durch einen Compilerschritt vorgenommen. Der zweite Schritt, die *Pfadselektion*, sucht nach Pfaden im CFG, welche sich für die Implementierung eignen.

Nach dem zweiten Schritt werden durch COMRADE alle Datenpfade für die HW-Kandidaten erzeugt. Der dritte Schritt der HW-Auswahl, die *Kandidatenbewertung*, beurteilt alle HW-Regionen durch den zu erwartenden Laufzeitgewinn gegenüber der SW-Realisierung und wählt somit aus den HW-Kandidaten die endgültigen HW-Realisierungen aus.

Grundlegend zum Verständnis der nachfolgenden Schritte ist die Sicht des Compilers auf das System aus Prozessor und RL. Oft wird der RL nur als Ergänzung zum Prozessor angesehen. Dadurch kann man die Bearbeitung der HW-Kandidaten durch den Compiler vereinfachen. Für COMRADE stellen Prozessor und RL gleichberechtigte Partner dar. Somit ist es auch für den RL möglich, vom Prozessor Teile eines Programms ausführen zu lassen, wenn diese nicht durch den RL bearbeitbar sind.

3.2.1 Schleifenduplikation

Für die Auswahl von HW-Regionen beschränken wir uns auf rechenintensive Schleifen. Nun lohnt es sich nicht immer, jede als geeignet ausgewählte Region komplett in HW auszuführen. Manchmal ist es sinnvoll, einzelne Pfade der Region für eine SW-Ausführung vorzusehen. Dieser Fall tritt z.B. ein, wenn auf einem Pfad der Region I/O-Anweisungen des Betriebssystems liegen, die nicht in HW ausgeführt werden können. Des Weiteren kann es während des Programmlaufs vorkommen, dass eine im Prinzip geeignete HW-Region wegen unpassender Eingabedaten in Einzelfällen nicht auf der HW ausgeführt werden kann. Darüber kann zur Übersetzungszeit aber nicht vorab entschieden werden.

Um die Entscheidung später treffen zu können, werden alle Kandidaten für eine HW-Ausführung im Compiler-Schritt Schleifenduplikation dupliziert. Dazu werden diese Bereiche im CFG verdoppelt (analog zu GarpCC [CaHW00]). Durch eine zusätzlich eingefügte Verzweigung kann auch zur Laufzeit noch zwischen HW- und SW-Ausführung gewählt werden. Dadurch ist es möglich, auf die ursprüngliche SW-Version zurückzugreifen, wenn sich während der Übersetzung oder Laufzeit zeigt, dass eine HW-Ausführung nachteilig ist. Die Behandlung dieser Ausnahme ist bisher noch nicht implementiert.

In COMRADE werden nicht nur innere Schleifen dupliziert. Somit kann man auch Beschleunigungspotential von äußeren Schleifen nutzen. Bei der Duplikation wird von den äußersten Schleifen an begonnen. Wurden diese in eine SW- und eine HW-Region dupliziert, so wird die SW-Region bearbeitet. Hierin wird nun die nächst innere Schleife dupliziert. Dieser Vorgang wiederholt sich, bis die innerste Schleife dupliziert wurde. Diese Vorgehensweise verhindert, dass zuerst von einer inneren Schleife HW- und SW-Version erzeugt und dann beim Duplizieren einer äußeren Schleife diese beiden Versionen dann in jeder der äußeren enthalten sind. Benötigt wird aber nur jeweils eine SW- und eine HW-Version einer Region.

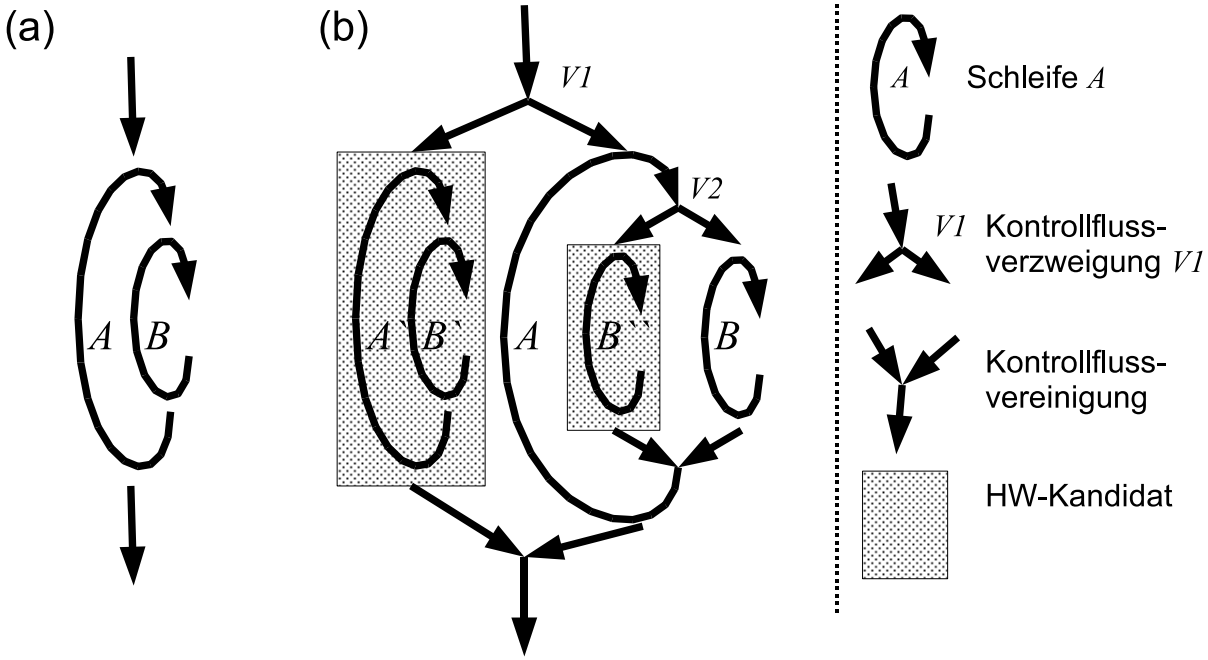


Bild 3.4 Schleifenduplikation schematisch: (a) Original, (b) Duplikationen

Ein Beispiel für die rekursive Duplikation zeigt schematisch Bild 3.4. Hier werden zwei ineinander geschachtelte Schleifen dupliziert. Man kann erkennen, dass nicht nur größtmögliche Regionen dupliziert werden. Mit den Bedingungen in den Verzweigungen im Kontrollfluss *V1* und *V2* kann zwischen drei Alternativen gewählt werden. So ergibt sich die Freiheit, die HW-Realisierung nur der inneren Schleife zu benutzen, wenn die zweite Region beispielsweise zu groß für den Ziel-RL ist. Die Regionen, welche durch das Duplizieren entstanden sind, werden **HW-Kandidaten** genannt. Sie besitzen genau einen Kontrollflussblock als Eingang, dessen Vorgänger nicht mehr der Region aber dem gesamten CFG angehört. Sie besitzen mindestens einen Kontrollflussblock, dessen Nachfolger nicht Teil der Region aber des CFG ist. Bei der Ausführung der Prozedur kann zwischen drei verschiedenen Versionen gewählt werden:

1. Werden auf Grund der Bedingungen in *V1* und *V2* jeweils die rechten Abzweige gewählt, so führt das Programm die Schleifen *A* und *B* aus, welche beide keine HW-Kandidaten sind.
2. Wird an der Kontrollflussverzweigung *V1* der rechte Zweig und an *V2* der linke Zweig gewählt, so werden die Schleifen *A* und *B``* und somit für die innere Schleife der HW-Kandidat ausgewählt.
3. Wird an der Kontrollflussverzweigung *V1* der linke Zweig gewählt, so werden die Schleifen *A`* und *B`* ausgeführt. Somit bilden beide Schleifen den HW-Kandidaten.

Während dieses Compiler-Schritts werden auch Kontrollflussblöcke in den CFG eingefügt, in denen zu einem späteren Zeitpunkt die Schnittstellen zwischen HW- und SW-Teilen der Applikation generiert werden. Diese Schnittstellen übertragen die zur Arbeit der HW benötigten Daten zur HW und wieder zurück zur SW.

Die Duplikation aller Schleifen in einem Programm würde die Laufzeit des Compilers stark erhöhen. Außerdem kann schon in diesem Schritt für viele Schleifen bestimmt werden, dass sie nicht in HW realisiert werden. Das erste Kriterium sind Profiling-Daten. Nur sehr oft ausgeführte Schleifen sind auch für eine HW-Ausführung lohnend. Wird eine Region selten ausgeführt, dann ist meist die Rekonfigurationszeit des RL größer als der Zeitgewinn der HW-Ausführung gegenüber der SW-Ausführung.

Ein weiteres Kriterium sind enthaltene Funktionsaufrufe. Wenn eine Schleife einen Funktionsaufruf enthält, so kann dieser durch COMRADE nicht direkt in HW realisiert werden. Ist die Definition der aufgerufenen Funktion dem Compiler bekannt (im Quelltext vorhanden), so kann sie an die aufrufende Stelle eingefügt werden (**Inlining**). Ein profil-basiertes Inlining (siehe Abschnitt 3.3) ermöglicht in diesem Zusammenhang, dass nur Funktionsdefinitionen in andere Funktionen eingefügt werden, wenn es sich auf Grund von Laufzeitinformationen wirklich lohnt. Durch das Inlining können größere Regionen für eine HW-Realisierung gefunden werden (Abschnitt 3.4).

3.2.2 Pfadselektion und Schnittstellengenerierung

Bei der Pfadselektion werden alle für eine eventuelle Ausführung in HW vorgesehenen, duplizierten Regionen des CFG bearbeitet. Während des Aufbaus eines Pfads (Weg durch den CFG) in einem HW-Kandidaten werden mögliche zugehörige Kontrollflussblöcke nach folgenden Auswahlkriterien betrachtet:

1. *Hardware-Ausführbarkeit*: Ein Kontrollflussblock kann nicht im Ziel-RL implementiert werden, wenn er Operatoren enthält, die nicht in HW ausführbar sind. Hierzu zählen zurzeit insbesondere I/O-Operationen und Funktionsaufrufe. Leere Kontrollflussblöcke sind auch auf dem Ziel-RL ausführbar, da sie die Ausführung nicht behindern. Sie tragen größtenteils nur Kontrollflussinformationen.
2. *Ressourcenverbrauch*: Der Ressourcenverbrauch aller gewählten Pfade in einem HW-Kandidaten darf die auf dem RL verfügbaren Ressourcen nicht überschreiten. Zum Zeitpunkt der Pfadselektion steht bisher nur der Ressourcenverbrauch der einzelnen

Operationen zur Verfügung. Deshalb wird von einem Zuwachs des Ressourcenverbrauchs von beispielsweise 40% von der vorliegenden Ressourcengröße bis zur fertigen Implementierung wegen noch fehlender Kontrollergenerierung ausgegangen. Diese Abschätzung ist so konservativ, dass hierbei keine HW-Kandidaten entstehen, welche mehr als die auf dem Ziel-RL verfügbaren Ressourcen verbrauchen.

3. *HW-Realisierbarkeit*: Ein Pfad eines HW-Kandidaten ist nicht realisierbar, wenn er nicht den Eingangsblock in den HW-Kandidaten und einen Ausgangsblock aus dem HW-Kandidaten enthält. Mit Rücksicht auf den Ressourcenverbrauch werden die Kontrollflussblöcke so ausgewählt, dass kurze Pfade mit dem dadurch wahrscheinlich geringem Ressourcenverbrauch ausgewählt werden. Dazu werden Pfade, die eine innere Schleife nicht betreten bzw. eine Schleife verlassen, bevorzugt.
4. *Ausführungsbeiwert*: Ein weiterer wichtiger Faktor für die Auswahl von Pfaden in den für HW vorgesehenen Regionen ist der Ausführungsbeiwert. Dieser ist wichtig, wenn der Verbrauch von Ressourcen eines HW-Kandidaten die verfügbaren Ressourcen des Ziel-RLs übersteigt. In diesem Fall sind nicht alle Operatoren in HW realisierbar und einige Kontrollflussblöcke müssen aus den Pfaden des HW-Kandidaten ausgeschlossen werden. Es sollten in diesem Fall nur die Pfade gewählt werden, welche einen hohen Laufzeitgewinn durch die HW-Realisierung versprechen. Das sind in erster Linie Pfade, auf denen Kontrollflussblöcke mit hohen Ausführungsbeiwerten liegen.

Ein weiteres wichtiges Kriterium für die Auswahl der Pfade ist der erzielte Laufzeitgewinn. Da sich dieser schwer aus den lokal verfügbaren Daten beim Betrachten einzelner Kontrollflussblöcke ableiten lässt, wird er in der Pfadselektion vernachlässigt. Im Schritt der Hardware-Entscheidung (3. Schritt der HW-Auswahl) hingegen ist die Entscheidung über Realisierung eines HW-Kandidaten vom erwarteten Laufzeitgewinn abhängig.

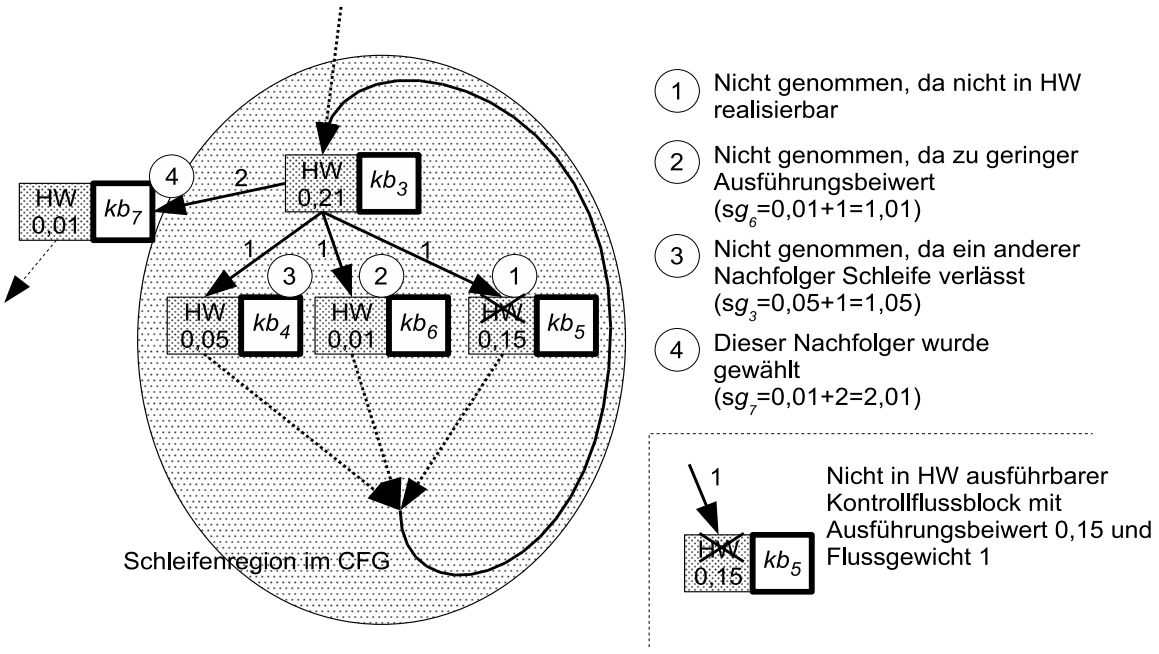


Bild 3.5 Kriterien für die Pfadselektion

Jeder HW-Kandidat hat neben mehreren Ausgangsblöcken nur einen Eingangsblock. Beim Bearbeiten eines Kandidaten wird dieser Kontrollflussblock für Kontrollflussblock beurteilt. Dabei werden die oben genannten Kriterien mit abfallender Priorität beachtet (Bild 3.5). Die Kriterien HW-Realisierbarkeit und Ausführungsbeiwert werden zu einem Gewicht zusammengefasst. Dieses **Selektionsgewicht** eines Kontrollflussblocks kb_n setzt sich aus $sg_n=ab_n+fg_n$ zusammen. Dabei berechnet sich das **Flussgewicht** fg_n aus:

$$fg_n=\begin{cases}2 \text{ wenn eine Schleife verlassen wird}\\0 \text{ wenn eine Schleife betreten wird}\\1 \text{ sonst}\end{cases}$$

Die Kriterien stellen sicher, dass als erstes alle Kontrollflussblöcke in der HW enthalten sind, welche für die Datenübergabe zwischen HW und SW nötig sind. Das sind die Eingangs- und Ausgangsblöcke eines HW-Kandidaten. Weiterhin wächst der in HW realisierte Teil während der Ausführung des Algorithmus in den am meisten verwendeten Blöcken einer Region.

Der Algorithmus beruht auf der rekursiven Ausführung des in Bild 3.6 dargestellten Programmflusses. Der Algorithmus beginnt am Eingangsblock eines HW-Kandidaten und sucht Pfade in diesem, indem er so lange nach den oben genannten Kriterien Nachfolger zum aktuellen Pfad hinzufügt, bis dieser an einem Ausgangsblock angelangt ist oder an einem Kontrollflussblock, der schon zu einem gewählten Pfad gehört. Dabei werden folgende Variablen in jedem Rekursionsschritt bearbeitet:

- *HwBlöcke* ist eine Menge von Kontrollflussblöcken, welche in den für die HW-Ausführung bestimmten Pfaden liegen.
- *NeueBlöcke* ist eine nach Priorität geordnete Liste von Kontrollflussblöcken. Hierin werden während der Pfadsuche alle Kontrollflussblöcke zwischengespeichert, welche aufgrund ihres Selektionsgewichts bisher nicht für einen Pfad ausgewählt wurden. Wurde ein gültiger aktueller Pfad gefunden und dieser der Menge *HwBlöcke* hinzugefügt, dann wird ein neuer Pfad mit dem Kontrollflussblock begonnen, welcher in dieser Liste die höchste Priorität hat.
- *Ressourcen* beinhaltet die Summe der Ressourcen aller in die Menge *HwBlöcke* aufgenommener Kontrollflussblöcke. Dadurch wird kontrolliert, ob die verfügbaren Ressourcen des Ziel-RLs nicht überschritten werden.

Dem rekursiven Algorithmus wird ein Kontrollflussblock übergeben. Ist dieser nicht in HW ausführbar, dann war die aktuelle Pfadsuche nicht erfolgreich (Schritt I) und dieses Ergebnis wird an den aufrufenden Rekursionsschritt zurückgegeben. Wenn der Kontrollflussblock in HW ausführbar ist, dann werden in Schritt II die Ressourcen des aktuellen Kontrollflussblocks zusammen mit den bisher gefundenen Kontrollflussblöcken und denen des Ziel-RLs verglichen. Bei zu großen Anforderungen ist auch der aktuelle Pfad nicht gültig.

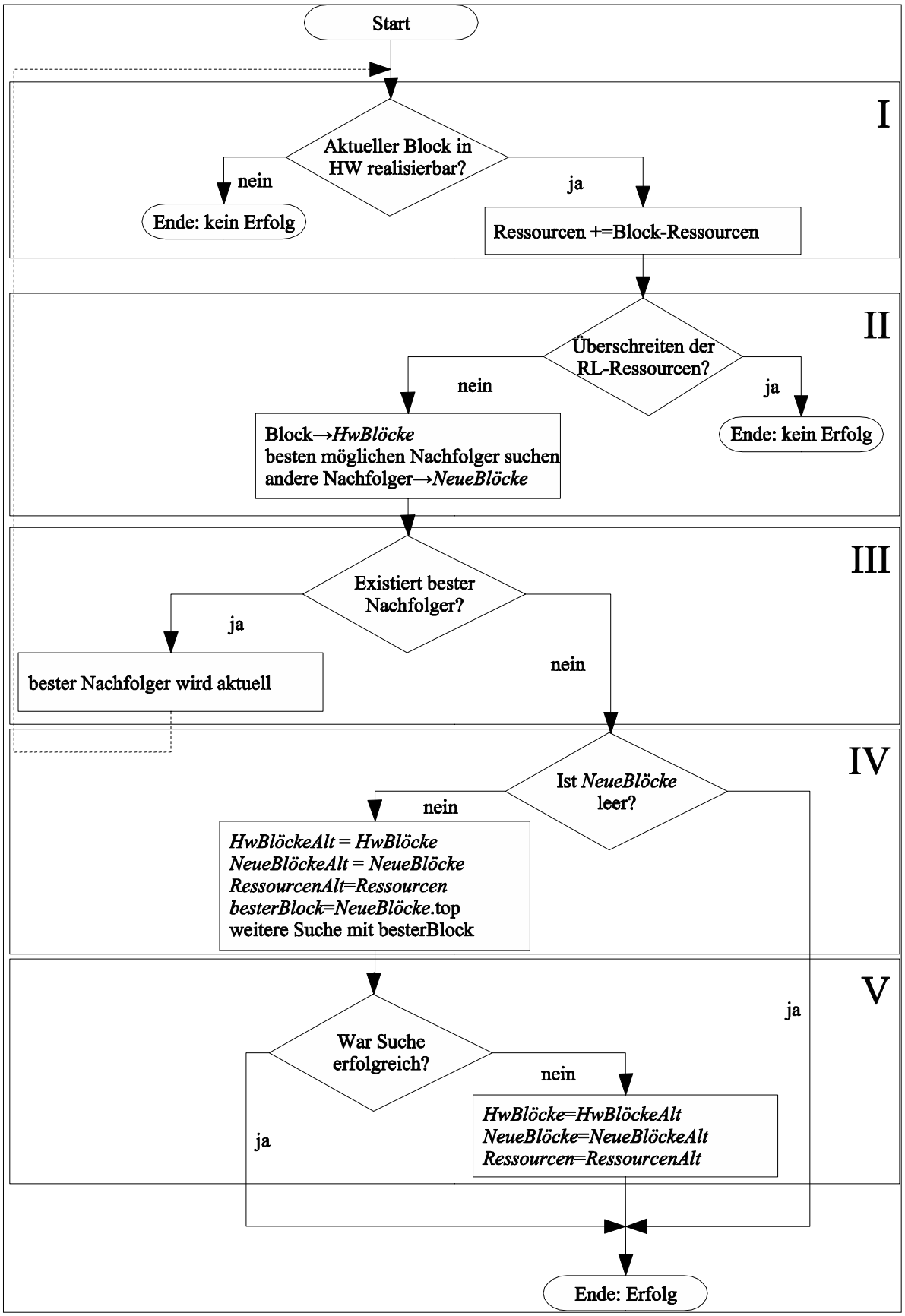
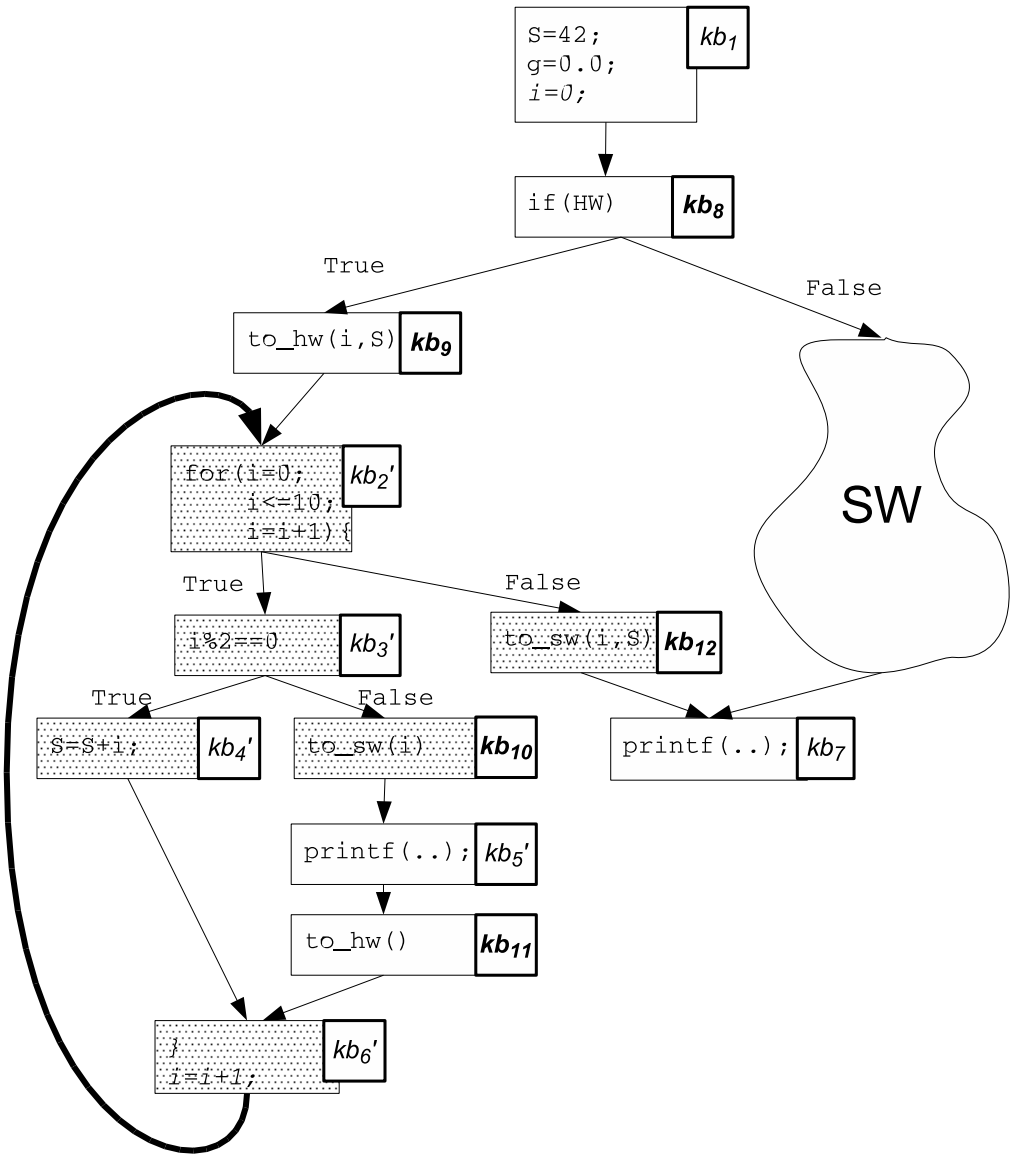


Bild 3.6 Algorithmus zur Pfadsuche

Werden durch die Hinzunahme des aktuellen Kontrollflussblocks die Ressourcen des RLs nicht überschritten, dann wird dieser Kontrollflussblock der Menge der in HW ausgeführten Kontrollflussblöcke hinzugefügt. Weiterhin werden alle Nachfolger des aktuellen Kontrollflussblocks betrachtet. Von jedem Nachfolger wird das Flussgewicht fg_n bestimmt. Wenn es mehr als einen Nachfolger gibt, wird der beste zum aktuellen Kontrollflussblock und alle anderen werden zwischengespeichert. Mit dem besten Kontrollflussblock wird nun ein neuer Rekursionsschritt begonnen und somit die Pfadsuche weitergeführt.

Während dieses Compiler-Schritts werden auch Kontrollflussblöcke in den CFG eingefügt, in denen zu einem späteren Zeitpunkt die Schnittstellen zwischen HW- und SW-Teilen der Applikation generiert werden. Diese Schnittstellen übertragen die zur Arbeit der HW benötigten Daten zur HW und wieder zurück zur SW.



Beispiel 3.7 Ausgesuchte Pfade

Da die Pfadsuche nicht zum Erfolg führen muss, werden der aktuelle Stand der Pfadsuche gesichert und die Pfadsuche mit dem neuen Nachfolger begonnen (Schritt V). Ist die Pfadsuche nicht erfolgreich, wird der gesicherte Stand wieder hergestellt. War die Pfadsuche erfolgreich, so wird dies an die aufrufende Prozedur zurückgegeben.

Nach der Pfadsuche werden Kontrollflussblöcke des HW-Kandidaten, welche nicht für die HW-Realisierung ausgesucht werden, zu Subregionen zusammengefasst und mit Kontrollflussblöcken für den HW-SW- und SW-HW-Übergang versehen. Der HW-Kandidat des Beispiel-CFG nach der Pfadselektion enthält nun als SW-Subregion den Knoten kb_5' (Beispiel 3.7). Neben den Eingangs- und Ausgangsblöcken kb_9 und kb_{12}, die auch Anweisungen zur Übergabe von Werten zwischen HW und SW enthalten, wurden die Knoten kb_{10} und kb_{11} eingefügt. Die Anweisung in kb_{10} übergibt den für die Ausführung der SW-Subregion benötigten Wert von *i*. Die Anweisung in kb_{11} signalisiert das Ende der Ausführung der SW-Subregion. Durch die Bedingung in der `if`-Anweisung in kb_8 kann entschieden werden, ob die HW- oder SW-Ausführung einer Region während des Programmlaufs stattfindet. Zum aktuellen Zeitpunkt der Bearbeitung ist der Wert der Bedingung aber noch nicht bestimmt, wird aber im Schritt der Hardware-Auswahl von COMRADE gesetzt.

Nachdem also die SW-Subregionen bestimmt wurden (Bild 3.8a), werden diese der SW hinzugefügt (Bild 3.8b). Der Prozessor führt diese Teile als Dienstleister für die HW aus. Der Vorteil liegt darin, dass keine komplizierten Verbindungen zu bestehenden SW-Regionen aufgebaut werden müssen.

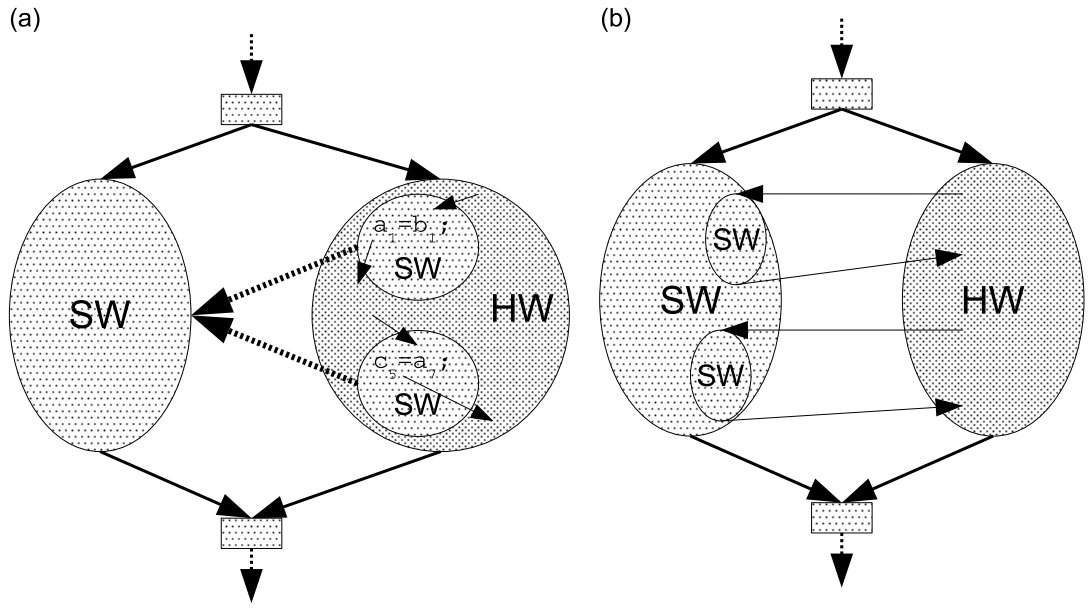


Bild 3.8 HW-Auswahl (a) und SW-Verschiebung (b)

Durch diese Vorgehensweise entstehen Datenabhängigkeiten zwischen Operationen in den HW-Kandidaten und in den neu erzeugten SW-Subregionen (im Beispiel durch Übergabe der Variablen *i*). Diese Abhängigkeiten beeinflussen die Laufzeit der HW-Region, da jeweils

ein Datenaustausch zwischen HW und SW stattfinden muss. Dieser Faktor beeinflusst wie auch der schon erwähnte Laufzeitgewinn die endgültige Realisierung einer Region in HW oder SW.

3.2.3 Hardware-Entscheidung

Der letzte Schritt der HW-SW-Aufteilung ist die Auswahl von HW-Kandidaten für die HW-Realisierung. Zwischen der Pfadselektion und der HW-Auswahl werden aus den CFGs der HW-Kandidaten jeweils Datenpfade erzeugt (Kapitel 4). Nun muss entschieden werden, ob die HW-Realisierung einen Geschwindigkeitsgewinn gegenüber der SW-Realisierung einer Region erbringt. Im Idealfall sollte man jede der beiden Realisierungen ausführen und die Ausführungszeiten messen.

Voraussetzung dazu ist die Implementierung der Datenpfade mit den sich an COMRADE anschließenden Synthesewerkzeugen für den Ziel-RL sowie die Messung von Laufzeiten der Regionen in SW und HW. Da jedoch die Erzeugung von HW-Implementationen zeitaufwendig ist und Mechanismen für die Messung nicht zur Verfügung standen, wird in COMRADE eine Abschätzung der Laufzeit angewendet. Diese wird für die SW und die HW auf Grund unterschiedlicher Voraussetzungen auch unterschiedlich bestimmt.

Für die Ausführung einer Region in SW wird davon ausgegangen, dass zur gleichen Zeit jeweils nur eine Operation durch den Prozessor ausgeführt werden kann. Die Ausführungszeit auf modernen Prozessorarchitekturen mit Pipelines und mehreren Ausführungseinheiten wird durch die Hinzunahme eines Faktors lf_{SW} angenähert. Dieser soll auch Effekte wie die Unterbrechung der SW-Ausführung durch das Betriebssystem ausgleichen. Um die Ausführungszeit eines Kontrollblocks zu bestimmen, benötigt man die der einzelnen ASTs der in diesem Kontrollflussblock enthaltenen Anweisungen. Jeder Ausdruck in diesem AST benötigt eine vom Prozessor abhängige Anzahl von Taktzyklen, in denen der während der Compilierung aus dem Ausdruck entstehende Maschinenbefehl (oder mehrere Maschinenbefehle) ausgeführt wird. Diese Anzahl von Taktzyklen wird jedem Ausdruck im AST zugeordnet.

Definition 14: Sei *Ausd* die Menge der in den ASTs einer Prozedur auftretenden Ausdrücke. *zykl*: $Ausd \rightarrow \mathbb{N}$ bildet jeden Ausdruck auf eine **Zyklenanzahl** ab.

Die Zyklenanzahl ist vom Prozessor abhängig und kann Tabellen des jeweils verwendeten Prozessors entnommen werden. Mit der Betriebsfrequenz eines Prozessors f_{Proz} kann somit die **Ausführungszeit eines Ausdrucks** mit:

$$ausf_{SW}(\, ausd_{n,j}\,) = zykl(\, ausd_{n,j}\,)/f_{Proz}$$

ermittelt werden. Darauf aufbauend ist die **Ausführungszeit einer Anweisung** *n*:

$$ausf_{SW}(\, n\,) = \sum_{j} ausfSW(\, ausd_{n,j}\,)$$

und verbunden damit die **globale Laufzeit einer Anweisung** *n*:

$$laufz_{SW}(\, n\,) = ausf_{SW}(\, n\,) * anz_{n}.$$

Die globale Laufzeit ist die Zeit, welche in einem bestimmten Teil eines Programms (hier eine Anweisung) während der Ausführung eines Programms schätzungsweise vergeht. Somit ergibt sich leicht die **globale Laufzeit eines Kontrollflussblocks** aus:

$$laufz_{SW}(kb) = \sum_{n \in kb} laufz_{SW}(n)$$

Die **globale Laufzeit einer SW-Region** *reg* ist damit mit dem Laufzeitfaktor lf_{SW} für die SW-Ausführung:

$$laufz_{SW}(reg) = \sum_{kb \in reg} laufz_{SW}(kb) * lf_{SW}$$

Die Laufzeit des aus einem HW-Kandidaten entstehenden Datenpfads muss anders berechnet werden. Auf dem Ziel-RL können die einzelnen HW-Module parallel und in Pipelines sowie spekulativ ausgeführt werden. Jede Ausführung eines Moduls im Datenpfad beginnt dann, wenn die dazu nötigen Daten vorhanden sind. An Multiplexern werden aus mehreren spekulativ berechneten Werten auf Grund von Kontrollflussinformationen die korrekten Daten ausgesucht. Dazu sind im Datenpfad Kanten eingefügt, welche die Kontrollflussabhängigkeiten darstellen. Ein Multiplexer kann nicht schalten, wenn diese Kontrollflussinformationen nicht vorhanden sind (Bild 3.9 und Kapitel 4).

Grundlegend zur Abschätzung der Ausführungszeit eines Datenpfads ist die **Ausführungszeit eines HW-Moduls** $ausf_{HW}(md)$. Diese wird wie in Abschnitt 3.1.2 durch einen FLAME-Zugriff ermittelt. Alle HW-Module besitzen einen kombinatorischen Pfad und ein abschließendes Register, damit die Implementierung auf dem Ziel-RL synchron ist. Somit benötigt man die **maximale Ausführungszeit aller Module** eines HW-Kandidaten

$ausf_{HW,max} = \max(ausf(md_1), ..., ausf(md_n))$.

Alle HW-Module werden so getaktet, dass diese maximale Ausführungszeit garantiert werden kann. Des Weiteren ist für jedes HW-Modul die Ausführungsanzahl der Anweisung *n*, in dem der zum HW-Modul gehörende Ausdruck enthalten ist, aus dem Profiling bekannt. Damit kann man **die globale Laufzeit eines HW-Moduls** aus

$laufz_{HW}(md) = ausf_{HW,max} * anz_n$ für $md = fl(ausd_{n,j})$

berechnen.

Grundlegend für die Gesamtlaufzeit eines Datenpfads sind die Laufzeiten der einzelnen Pfade im Datenpfad von einem Eingangsregister zu einem Ausgangsregister eines HW-Kandidaten. Auf diesem als **Modulpfad** genannten Pfad können Module des Datenpfads, Kontrollflussblöcke einer SW-Subregion und die Übergänge zwischen beiden enthalten sein. Die Modulpfade können auch Schleifen enthalten. In diesen Schleifen können so die Knoten auch mehrmals auftreten. Somit muss es erlaubt sein, dass in diesem Datenpfad jeder Knoten zweimal vorhanden sein kann.

Die Laufzeit eines Pfads *pf* in einer SW-Subregion ist ähnlich zur Laufzeit einer SW-Region die Summe der Laufzeiten aller Kontrollflussblöcke, welche sich auf diesem Pfad befinden:

$$laufz_{SW}(pf) = \sum_{kb \in pf} laufz_{SW}(kb)$$

Sind auf dem Modulpfad *r* Kommunikationsverbindungen zwischen der SW und HW notwendig, um Daten auszutauschen (Variablenübergabe), dann ergibt sich die geschätzte Laufzeit eines Modulpfads *mp* mit *m* HW-Modulen aus:

$$laufz_{HW}(mp) = \sum_{md \in mf} laufz_{HW}(md) + laufz_{SW}(pf) + r * t_K$$

Jeder mögliche Modulpfad für einen Datenpfad wird nun aufgestellt und der längste wird zur Berechnung zur Laufzeit eines Datenpfads verwendet. Die Laufzeit eines Datenpfads ergibt sich aus der maximalen Laufzeit aller Modulpfade eines Datenpfads *df*, der Rekonfigurationszeit und dem Laufzeitfaktor lf_{HW} zu:

$laufz_{HW}(df) = lf_{HW} * (\max(laufz_{HW}(mp_1), ..., laufz_{HW}(mp_n)) + t_R)$ mit $mp_k \in df$

Der Laufzeitfaktor soll die in der HW-Ausführung auftretenden Effekte durch spekulative Ausführung und Pipelining ausgleichen. Wegen des Pipelinings ist der Laufzeitfaktor lf_{HW} abhängig von der Anzahl der Module im Modulpfad. Die Anzahl der auf einem Modulpfad enthaltenen Module geht umgekehrt proportional in den Laufzeitfaktor ein.

Stehen die Laufzeiten der HW- und SW-Realisierung fest, dann kann auf Grund dieser Daten entschieden werden, ob die HW-Implementation gewählt wird. Die entscheidende Verzweigung im Anwenderprogramm wird dann auf den entsprechenden Wert gesetzt.

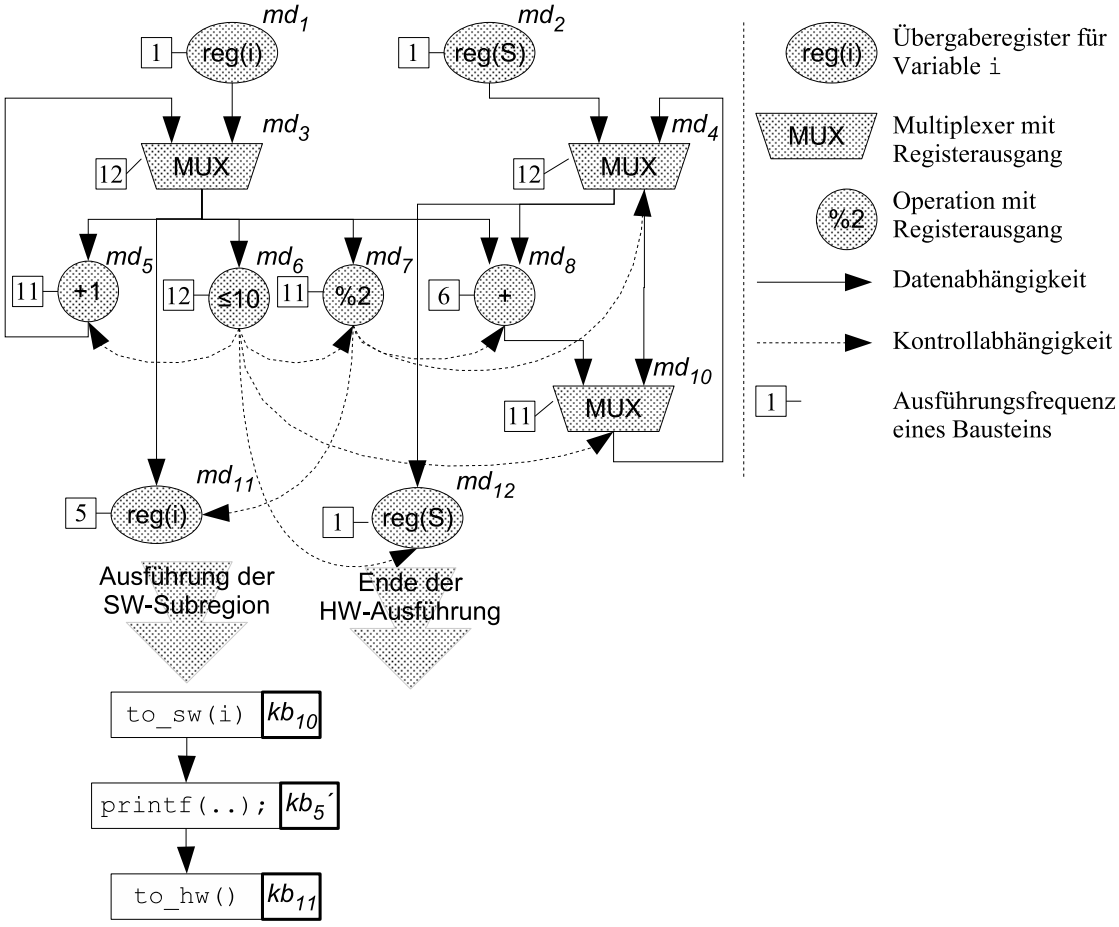


Bild 3.9 DFG des Beispiels

Am Datenpfad des Beispiels (Bild 3.9) kann man die Übergänge zwischen der HW und SW an den Registern für die Variablen *i* und *S* erkennen. Hier haben wir die besondere Situation, dass keine Daten von der Ausführung der SW-Subregion wieder in die HW zurückgegeben werden. Nur der Wert von *i* wird in die SW-Subregion übergeben. Der längste, nur aus HW-Modulen bestehende Pfad besteht aus den Modulen md_1, md_3, md_5, md_3, md_6, md_7, md_8, md_{10}, md_4, md_8, md_{10}, md_4, md_{12}. Beispielhaft wurde für diese Module aus GLACE ein Gesamtverbrauch von 116 Taktzyklen ermittelt. Mit einer maximalen Laufzeit von *l* Sekunden für ein HW-Modul und der Länge des längsten Modulpfads von 13 Modulen (Laufzeitfaktor 1/13) ergibt sich somit eine geschätzte Laufzeit von:

$$lauf_{HW}(df) = 116*1/13*l + t_R + 2*t_K$$

Vergleicht man diese Laufzeit mit der geschätzten Laufzeit der SW-Implementierung in Abhängigkeit mit der Taktfrequenz des verwendeten Prozessors (Bild 3.10), so kann man leicht erkennen, wann sich eine HW-Implementierung lohnt.

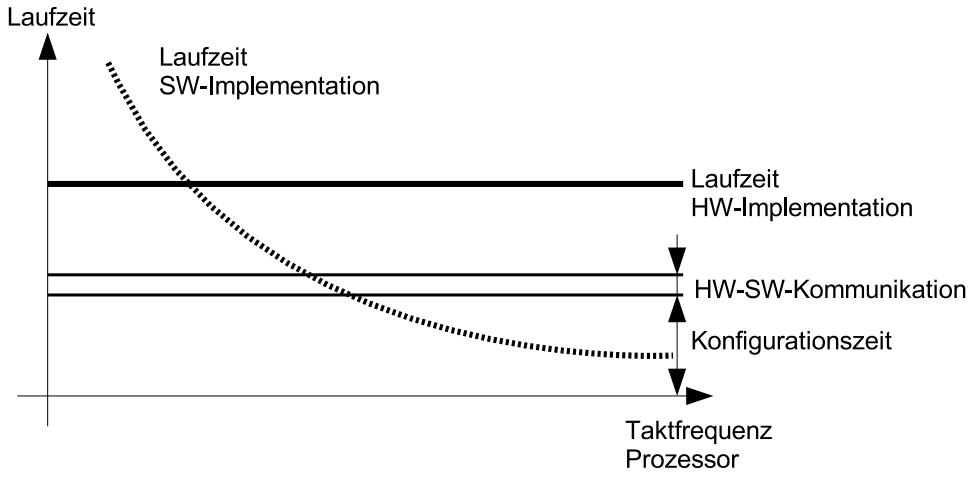


Bild 3.10 Laufzeitentwicklung von HW- und SW-Implementierung

3.3 Optimierungen der Hardware-Größe

Bei der Betrachtung der entstehenden Datenpfade bei Benchmark-Programmen aus den Bereichen Kompression und Dekompression sowie Ver- und Entschlüsselung war zu erkennen, dass viele der entstandenen Datenpfade nur einen geringen Teil der Ressourcen des Ziel-RLs benutzen. Bei näherer Betrachtung war zu erkennen, dass dies seinen Grund in der guten Strukturierung der Anwenderprogramme hatte. Es war für den Algorithmus so nicht möglich, mehrere geschachtelte Schleifen zu bearbeiten, da die Pfadselektion nicht Prozedurübergreifend arbeitet und Schleifenschachtelungen oft auf mehrere Prozeduren aufgeteilt waren.

Eine Abhilfe ist hierbei die Verwendung von *Procedure-Inlining*, welches vorhandene Prozedurdefinitionen an die Stellen im Anwendungsprogramm einfügen, an denen sie aufgerufen werden. Dadurch kann man erreichen, dass auch mehrere geschachtelte Schleifen durch

COMRADE bearbeitet werden können. Führt man das Inlining aber ohne Beschränkungen durch, so kann es bei großen Programmen leicht zu einer Quellcode-Explosion kommen, da man alle Prozeduren in die Prozedur `main` einfügen würde. Das Einfügen der Prozeduren muss also ähnlich wie in [Call02] beschränkt werden (profil-basiert). Durch die Unterstützung geschachtelter Schleifen brauchen in COMRADE demgegenüber aber nur folgende Kriterien beachtet zu werden:

1. Die einzufügende Prozedur darf keinen Aufruf einer Prozedur enthalten, die auch als Quellcode vorliegt. Die aufgerufene Prozedur würde durch COMRADE in jedem Fall auch durch die HW-Auswahl bearbeitet. Deshalb werden Schleifen, welche einen Aufruf zu solch einer Prozedur enthalten, nicht bearbeitet. Es könnten Aufrufe einer HW-Implementation aus einer anderen HW-Implementation entstehen. Diese verbrauchen Zeit und wurden in COMRADE durch das Zulassen von mehreren ineinander geschachtelten Schleifen für die HW-Auswahl umgangen.
2. Die Anweisung, welche den Aufruf der einzufügenden Prozedur darstellt, muss einen Ausführungsbeiwert über einem gegebenen Faktor besitzen. Hiermit wird der Aufwand für die HW-Erstellung reduziert.

Die Auswirkungen des Inlinings auf die Größe der Datenpfade und die Wirksamkeit werden im nächsten Abschnitt diskutiert.

3.4 Praktische Ergebnisse

In diesem Abschnitt wird die Anwendbarkeit der in diesem Kapitel eingeführten Algorithmen in praktischen Anwendungen belegt. Die Daten vervollständigen die theoretische Beschreibung der HW-Auswahl anhand verschiedener Anwendungsprogramme. Grundlegend hierfür sind Daten, welche aus GLACE für einen FPGA Xilinx Virtex 1000 ermittelt wurden.

In Tabelle 3.11 werden zu jedem Anwendungsprogramm die als HW-Kandidaten ausgesuchten Regionen genauer spezifiziert. Das Programm `versatility` ist ein Bildkompressionsverfahren. Durch `adpcm` werden Audio-Daten komprimiert. In `capacity` wird eine Datei erzeugt, welche durch den Huffman-Kompressionsalgorithmus die Kapazität von Architekturen ermittelt. Das Programm `g721` komprimiert Sprache und `pegwit` implementiert eine Public-Key-Dateikodierung und -dekodierung.

Die Daten wurden nach dem Schritt der Pfadselektion gemessen. Dazu werden in den Spalten zwei bis vier diese Regionen über die in ihnen enthaltene Anzahl von Schleifenköpfen klassifiziert. Es wird jeweils die Anzahl der als HW-Kandidaten ausgewählten Regionen den insgesamt vorhandenen Regionen gegenübergestellt. Die Anzahl der Schleifen ist natürlich größer als die reale Anzahl von Schleifen im jeweiligen Programm. Das kommt daher, dass nicht nur innere sondern auch umschließende Schleifen inklusive der inneren bei der Duplikation verdoppelt werden.

In der fünften Spalte kann die prozentuale Anzahl von nach der Pfadselektion in HW verbliebenen CFG-Blöcken entnommen werden. Spalte sechs gibt die von den HW-Kandidaten verwendeten HW-Ressourcen auf dem Ziel-RL an. Die letzte Spalte zeigt die

durch die Auswahl von SW-Subregionen unterbrochenen Datenabhängigkeiten an. Sind diese zu hoch, lohnt sich wegen der Kommunikationszeiten zwischen HW und SW keine Realisierung in HW.

Programm	Regionen mit 1 Schleife	Regionen mit 2 Schleifen	Regionen mit >2 Schleifen	#HW-Blöcke in Kand.	Ressourcen in Kand. (CLBs)			Unterbrochene Datenabh.
	HW-Kand./ Total	HW-Kand./ Total	HW-Kand./ Total	∅	min	∅	max	#Var./ #Transf.
versatility	9/15	2/4	0/1	100%	544	1454	3129	0/0
adpcm	1/1	0/0	0/0	100%	1395	1395	1395	0/0
capacity	16/17	3/3	1/2	85,7%	38	188,9	566	12/1600
g721	7/10	0/0	0/0	100%	97	910	2041	0/0
pegwit	43/73	4/9	0/1	100%	186	823	2116	0/0

Tabelle 3.11 Erzeugte HW-Kandidaten und unterbrochene HW-/SW-Datenabhängigkeiten

Aus der Tabelle ist beispielsweise ablesbar, dass `versatility` 15 innere Schleifen enthält, von denen neun als HW-Kandidaten ausgewählt wurden. Außerdem gibt es vier Schleifenschachtelungen mit zwei sowie eine mit mehr als zwei Schleifen, von denen zwei bzw. keine für einen HW-Kandidaten vorgesehen werden. Von allen in den HW-Kandidaten enthaltenen Kontrollflussblöcken könnten alle in HW übersetzt werden, da die darin verwendeten Operatoren auch in HW ausführbar sind. Die dabei erzeugten HW-Implementierungen verwenden geschätzt minimal 473CLBs, maximal 3064CLBs und im Durchschnitt 1398CLBs. Da die HW-Kandidaten keine SW-Subregionen enthalten, entstehen auch keine Variablen-Transfers zwischen der HW-Implementierung und den SW-Subregionen.

Erkennbar ist, dass ein Großteil der Schleifen in den Anwendungsprogrammen als HW-Kandidat in Frage kommt. Außerdem sind die in ihnen vorhandenen Operationen fast vollständig in HW ausführbar. Auch die Kommunikation zwischen HW und SW aufgrund von unterbrochenen Datenabhängigkeiten stellt in den meisten Fällen keinen limitierenden Faktor für die HW-Implementation dar. Somit kann hier nur noch die Laufzeit gegen eine HW-Implementierung sprechen.

Weiterhin kann man an der Größe für die HW-Kandidaten erkennen, dass jeder einzelne von ihnen den Ziel-RL mit einem Ressourcenangebot von 6144 CLBs nicht beansprucht. Deswegen wurde versucht, die HW-Kandidaten durch das in Abschnitt 3.3 beschriebene Verfahren des profil-basierten Inlinings zu vergrößern. Wenn man die Duplikation der Schlei-

fen nicht profil-basiert ausführt, dann ist eine Vergrößerung der HW-Kandidaten möglich [KaKo04]. Werden dagegen nur lohnende Schleifen dupliziert, können nur teilweise Vergrößerungen der HW-Kandidaten mehr festgestellt werden (capacity in Tabelle 3.12). Das Problem ist in der hohen Rekonfigurationszeit für aktuelle FPGAs zu suchen. Diese vergrößern die Laufzeit jedes einzelnen HW-Kandidaten derart, dass es sich nicht lohnt, Schleifen mit einem geringeren Ausführungsbeiwert zu duplizieren. Dadurch lohnt sich auch nicht das Inlining von Funktionen mit einem höheren Ausführungsbeiwert. Bei einem RL mit schnelleren Rekonfigurationszeiten ist somit mit einer Verbesserung der Ausnutzung durch das Inlining zu rechnen.

Programm/ eingebundene Funktionen	Schwell-wert	Regionen mit 1 Schleife	Regionen mit 2 Schleifen	Regionen mit >2 Schleifen	Ressourcen in Kand. (CLBs)			Unterbrochene Datenabh.
		HW-Kand./ Total	HW-Kand./ Total	HW-Kand./ Total	min	∅	max	#Var./ #Transf.
versatility/2	0.01	11/15	4/4	1/2	221	1188	3141	0
capacity/1	0.01	16/17	3/3	1/2	40	277	880	19/41262

Tabelle 3.12 Eingebundene Schleifen und veränderte Ressourcen

Ein anderer Ansatz zur besseren Ausnutzung des Ziel-RLs liegt im Zusammenfassen von HW-Implementierungen zu einer Konfiguration. Diese Optimierung wird in (Kapitel 6) beschrieben.

Programm	HW-Impl./HW-Kand mit 1 Schleife	HW-Impl./HW-Kand. mit >1 Schleife	Geschätzter Beschleunigungsfaktor	Ressourcen in Impl. (CLBs) min / ∅ / max		
versatility	8/9	1/2	13,8-63,6	456	1273	3064
adpcm	1/1	0/0	13,2	1346	1346	1346
capacity	5/16	0/4	1,3-6,3	32	137,6	561
g721	2/7	0/0	22,6-116,9	592	1304	2016
pegwit	11/43	0/4	8,3-78,6	181	716	2147

Tabelle 3.13 Ausgewählte Regionen für HW-Beschleunigung

Tabelle 3.13 zeigt die Auswahl von HW-Kandidaten für die HW-Implementierung. Für jedes Beispielprogramm sind in den Spalten zwei und drei die HW-Implementierungen mit einer vorausgesagten Beschleunigung der als HW-Kandidaten gewählten Regionen mit einer Schleife gegenübergestellt. Die Spalte vier enthält die Minimal- und Maximalwerte der geschätzten Beschleunigungsfaktoren (siehe Abschnitt 3.2.3) einer HW- gegenüber einer SW-Implementierung. Die fünfte Spalte zeigt die in den HW-Implementierungen verbrauchten Ressourcen.

Für die Schätzungen wurde der 100MHz–Prozessor und der Virtex-FPGA des ACE-2-ACS zugrunde gelegt. Die erhaltenen Werte zeigen, dass in vielen Beispielen mit einer HW-Implementierung eine Beschleunigung gegenüber der SW-Implementierung erreicht werden kann, auch wenn die Zeit zur Rekonfiguration des Ziel-RLs mit zur HW-Ausführungszeit gerechnet wird. Außerdem kann man erkennen, dass in den gewählten Beispielen nur eine Region mit mehr als einer Schleife für die HW-Implementierung gewählt wurde. Das Beschleunigungspotential scheint auch in diesem Fall den Konfigurationsaufwand nicht zu rechtfertigen.

3.5 Erweiterungsmöglichkeiten

Die Experimente zur HW-SW-Auswahl zeigten, dass das Verfahren bei den getesteten Programmen gut anwendbar ist. Schwächen zeigt sie noch bei der Abschätzung der HW-Ausführung. Diese ist nicht mit einer großen Anzahl verschiedenartiger HW-Implementierungen getestet worden. Als Erweiterung dieser Abschätzung sollte dies nachgeholt und die daraus erzielten Ergebnisse in die Implementierung von COMRADE eingefügt werden (CE_00). Eine Verbesserung könnte es beispielsweise sein, die starre Eigenschaft eines Modulpfads, dass jedes HW-Modul höchstens zweimal auftreten darf, flexibler zu gestalten. Eine Annahme wäre beispielsweise, dass die Anzahl des Auftretens eines HW-Moduls in einem Modulpfad gleich der Anzahl der Dateneingänge ist (CE_01). Da es aber nicht viele HW-Module mit mehr als zwei Dateneingängen gibt, wird dadurch die Abschätzung nur minimal verbessert. Ansonsten ist auch denkbar, dass in die Abschätzung der Typ der Kanten mit einfließt. Eine Möglichkeit besteht im Ignorieren von Kontrollabhängigkeiten.

Weiterhin sollte sich das in Abschnitt 3.1.1 angesprochene statische Profiling besser für den Erhalt von Profiling-Daten eignen als das bisher verwendete dynamische Profiling (CE_02). Der hauptsächliche Vorteil liegt darin begründet, dass ein Programm zum Erhalt der Daten nicht mehr ausgeführt werden muss. Das spart Zeit beim Compilieren, da auch das Abgleichen der während der Programmausführung erhaltenen Laufzeitinformationen mit der aktuellen Zwischendarstellung entfällt. In diesem Fall können auch Programme bearbeitet werden, welche ohne Unterbrechung laufen, mit dem dynamischen Profiling also nicht bearbeitbar sind.

Kapitel 4

Datenpfaderzeugung aus HW-Kandidaten

Ein wichtiger Bestandteil von COMRADE ist die Erzeugung von Datenpfadbeschreibungen aus den HW-Kandidaten zur weiteren Bearbeitung durch Platzierungs- und Verdrahtungswerkzeuge. Dieses Kapitel beschreibt die Erstellung von Datenflussgraphen aus den HW-Kandidaten und eines steuernden Kontrollers, welcher die Ausführung des Datenflussgraphen als Pipeline und teilweise spekulativ zulässt.

Der Kontroller entsteht dabei nicht aus einem statischen Scheduling, bei dem die Ausführungsreihenfolge der HW-Module schon vollständig während des Compilier-Prozesses feststeht. Er unterscheidet sich somit von Verfahren wie den klassischen Scheduling-Verfahren wie Modulo-Scheduling [MeEA03] oder Software-Pipelining [AJLA95, AkJa02] und deren Erweiterungen [ShMo98, Petk01, MaRi01, JuHW94], dem Compile-Zeit-Scheduling von HW-Modulen mit variabler Laufzeit [HaLe97, RaRL00] oder dem Pipeline Vectorization [Wein01]. Außerdem wird durch die Art der Ausführung des DFG eine spekulative Ausführung unterstützt [HoEr95, LaRJ98].

4.1 Schematischer Aufbau eines Datenpfads

Die Erzeugung eines Datenpfads in COMRADE erfolgt in zwei Schritten. Im ersten wird der Datenflussgraph (DFG) erzeugt, welcher alle Daten- und Kontrollabhängigkeiten zwischen den einzelnen HW-Modulen eines HW-Kandidaten enthält. Im zweiten Schritt werden diese Abhängigkeiten für die Erzeugung eines Kontrollers verwendet, welcher die Weitergabe von Werten zwischen den HW-Modulen steuert (Bild 4.1). Man kann erkennen, dass der DFG-Teil des Datenpfads Daten aus dem SW-Teil eines Anwendungsprogramms erhält und diese verarbeitet. Bei der Verarbeitung werden Informationen aus HW-Modulen, die Kontrollinformationen berechnen, an den Kontroller weitergegeben. Am Ende der Berechnungen gibt der Datenpfad Ergebnisse an den SW-Teil zurück.

Der Kontroller wiederum erhält Kontrollinformationen aus dem SW-Teil einer Anwendung (Start der HW-Ausführung, Weiterführen der HW-Ausführung) und berechnet durch diese und die aus dem DFG erhaltenen Kontrollinformationen seinen internen Zustand. Aus diesen werden Steuersignale für den DFG (enable-Signale der Register, Auswahl von Multiplexern) erzeugt. Erkennt der Kontroller das Ende der HW-Ausführung oder eine Unterbrechung, werden diese Informationen an den SW-Teil übergeben.

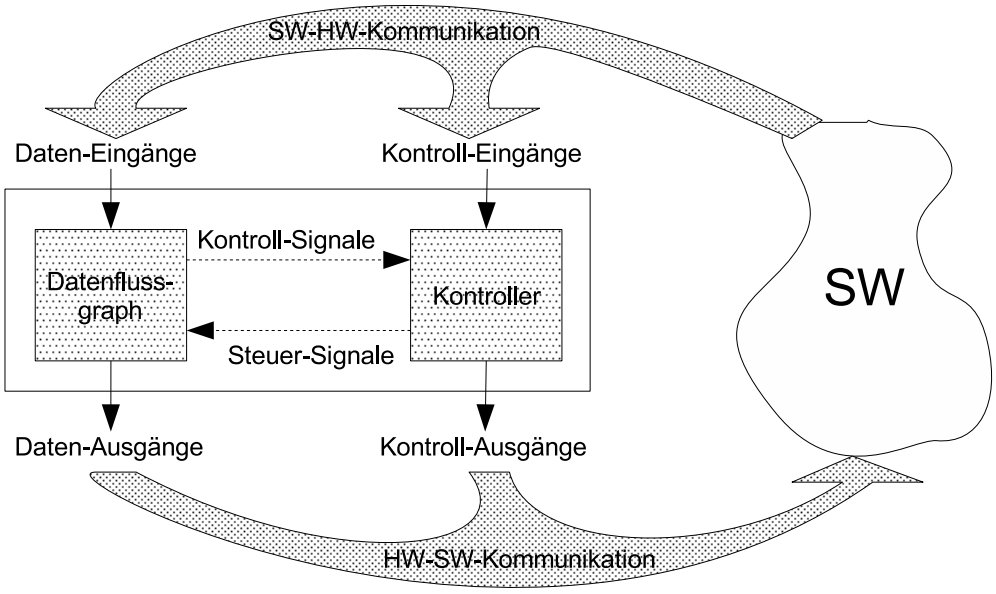


Bild 4.1 DFG und Kontroller bilden einen Datenpfad [BoDP98]

4.2 Erzeugung des Datenflussgraphen

Dieser Abschnitt beschreibt das Erstellen des Datenflussgraphen aus den HW-Kandidaten. Nach einer Einführung in die in COMRADE verwendete SSA-Form wird die Konvertierung der HW-Kandidaten in einen DFG beschrieben. Dieser ist die Grundlage für die Erzeugung des Kontrollers, welcher die Ausführung des DFG auf dem Ziel-RL steuert.

4.2.1 Aufbau des Datenflussgraphen

Die von COMRADE verwendeten DFGs können verschiedene Typen von HW-Modulen enthalten (Bild 4.2). Das sind im Einzelnen:

1. *HW-Module mit konstanter Laufzeit* (Bild 4.2a): Die HW-Module mit konstanter Laufzeit bestehen aus kombinatorischer Logik. Darüber hinaus kann am Ausgang dieser Logik auch noch ein Register vorhanden sein, um das Ergebnis der Berechnung zu speichern. Die Laufzeit durch die Logik dieser Module wird als konstant angenommen. Eine spezielle Art dieser HW-Module sind die für die Ergebnisauswahl verwendeten Multiplexer. Die Bitbreite des Auswahl-Eingangs ist gleich der Anzahl der Eingänge. Jedes Bit des Auswahl-Eingangs steuert somit das Durchschalten eines Eingangs zum Ausgang des Multiplexers. Die Steuerung des Auswahl-Eingangs wird vom Kontroller übernommen.
2. *HW-Module mit variabler Laufzeit* (Bild 4.2b): Die Bearbeitung von Daten in HW-Modulen mit variabler Laufzeit wird durch ein Signal gestartet. Nach einer datenabhängigen Anzahl von Taktzyklen steht ein Ergebnis bereit. Dies wird durch einen Ausgang signalisiert. Diese HW-Module besitzen eine minimale und eine maximale Laufzeit. Eine wichtige Untergruppe von HW-Modulen mit variabler Laufzeit sind Speicherzugriffe. Da-

bei werden zurzeit nur Ramdom-Access-Zugriffe unterstützt. Eine Erweiterung auf Streaming-Zugriffe [Call02] ist einfach realisierbar.

3. *HW-Module mit Pipeline-Stufen* (Bild 4.2c): Bei HW-Modulen mit eingebauten Pipelines ist es möglich, mehrere Datensätze zur gleichen Zeit zu bearbeiten. Diese Module zeigen an, wenn sie einen neuen Datensatz annehmen können. Ist dieser vorhanden, wird dies dem HW-Modul mitgeteilt. Am Ausgang des HW-Moduls werden auch Signale für die Fertigstellung einer Berechnung und die erfolgte Übernahme durch ein anschließendes HW-Modul ausgetauscht. Diese Art der HW-Module wird zurzeit noch nicht vollständig in COMRADE integriert. Das Pipelining innerhalb der Module wird nicht ausgenutzt. Sie werden als HW-Module mit variabler Laufzeit angesehen.

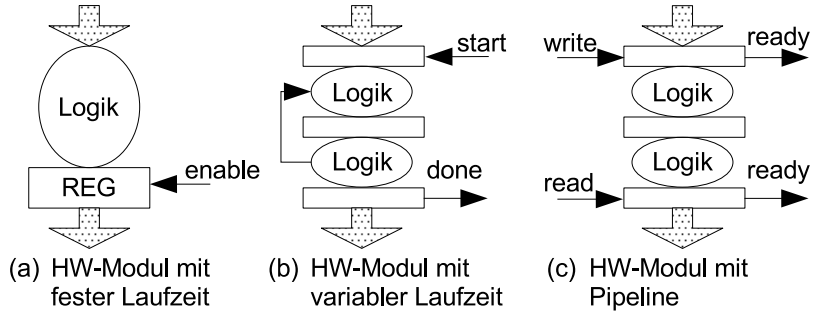


Bild 4.2 Typen von HW-Modulen

Für das Lesen von Daten vom SW-Teil und für das Schreiben von Daten zum SW-Teil enthält der DFG spezielle Register, deren reale Implementierung von der eigentlichen Kommunikationsstruktur zwischen dem Prozessor und dem RL abhängig ist. Weiterhin bilden alle Ausgänge von HW-Modulen, welche Kontrollinformationen erzeugen, Eingänge für den Kontroller. Alle am Beginn einer HW-Ausführung zu startenden HW-Module (Eingaberegister, Konstanten) werden in einer Liste gespeichert. Diese wird für die Generierung des Kontrollers verwendet.

4.2.2 Darstellung eines CFG in SSA-Form

Um den Datenflussgraph aus einem CFG zu erstellen, ist es notwendig, die Datenabhängigkeiten in diesem herauszufinden. Zu diesem Zweck stehen mehrere Möglichkeiten zur Verfügung. Die Verkettung von Schreibzugriffen auf Variablen und Lesezugriffen von diesen (*Def-Use-Chains*) hatte für die Benutzung in COMRADE zu viele Nachteile [KKGR03]. So ist eine Verwaltung der Def-Use-Chains nur parallel zur eigentlichen Programmdarstellung möglich. Weiterhin wäre bei Optimierungen stets eine Änderung beziehungsweise ein Neuaufbau der Def-Use-Chains erforderlich. Der dazu notwendige Verwaltungsaufwand und die im Compiler verbrauchte Rechenzeit waren nicht tragbar. Deshalb stützt sich COMRADE wie auch andere Projekte [Call02, BuGo02] auf eine spezielle Darstellung eines CFG – die **Static-Single-Assignment-Form** (SSA-Form).

Die heraustretende Eigenschaft der SSA-Form besteht darin, dass für jede Variable nur eine Stelle im Programm existiert, an der diese beschrieben wird (Single-Assignment). Damit sich aber die ursprüngliche Semantik eines Programms nicht verändert, werden in Kontrollflussblöcken mit mehreren Vorgängern bei Bedarf so genannte Φ-Anweisungen eingefügt. Diese weisen in Abhängigkeit vom vergangenen Kontrollfluss Werte an eine Variable zu [Appe98].

Die SSA-Form wurde ursprünglich für die Behandlung von skalaren Variablen entwickelt. Da aber in realen Programmen auch Speicherzugriffe auftreten, müssen diese auch behandelt werden. Für diese Erweiterungen wurden Ideen der Array-SSA-Form [KnSa98, FiKS00] aufgegriffen und an die Bedürfnisse in COMRADE angepasst. Ein großer Unterschied besteht darin, dass die erwähnten SSA-Formen Speicherzugriffe nur auf Anweisungsebene behandeln. Tritt mehr als ein Speicherzugriff in einer Anweisung auf, können diese durch die SSA-Formen nicht mehr ausreichend behandelt werden. Wir benötigen aber auch das Aufstellen einer Reihenfolge von Speicherzugriffen innerhalb einer Anweisung. Die SSA-Form muss also für Speicherzugriffe auch innerhalb von Anweisungen unterschiedliche Variablenzugriffe generieren können. Deshalb wird bei uns jeder Speicherzugriff separat behandelt.

Eine weitere notwendige Eigenschaft der Behandlung von Speicherzugriffen ist die Festlegung einer Reihenfolge von diesen Zugriffen durch die aufgebauten Abhängigkeiten zwischen den Variablen. Zu diesem Zweck wird jeder Speicherzugriff so behandelt, dass er eine alte virtuelle Variable liest und eine neue erzeugt. Der Vorteil liegt darin begründet, dass alle Speicherzugriffe auch dann noch in einer richtigen Reihenfolge ausgeführt werden, wenn Speicherzugriffsanalysen keine Abhängigkeiten oder Unabhängigkeiten zwischen den Zugriffen ermittelt haben. Somit werden also zuerst alle Speicherzugriffe (Felder, Pointer) so aufgefasst, dass sie sich gegenseitig beeinflussen könnten. Diese Abhängigkeiten können dann aber durch geeignete Analysen (Abschnitt 5.1) aufgehoben werden. Damit sind dann auch parallele Zugriffe auf den Speicher möglich, wenn es der Ziel-RL erlaubt.

```
int main() {
  int i;
  int a[10];
  a[0]=1;
  a[1]=1;
  for( i=2; i<10; i=i+1 ) {
    if( i%2 == 0 ) {
      a[i]=a[i-1]+a[i-2];
    } else {
      a[i]=0;
    }
  }
  printf( "Ergebnis %i\n", a[9] );
}
```

Beispiel 4.3 for-Schleife mit Speicherzugriffen

In der SSA-Form zum Beispiel 4.3 werden für die Variable `i` die Variablen `i0`, `i1` und `i2` erzeugt. Für die zusammentreffenden Definitionen von `i` im Kontrollflussblock kb_2 aus den Kontrollflussblöcken kb_1 (`i0`) und kb_6 (`i2`) wird eine Φ-Anweisung eingefügt, welche je nach Vergangenheit der Programmausführung eine der Variablen an `i1` zuweist (Bild 4.4). Zur Erleichterung der Darstellung wurde die `for`-Anweisung in ihre Bestandteile zerlegt.

Zugriffe auf den Speicher werden in der SSA-Form von COMRADE anders dargestellt. Jeder Speicherzugriff (Lesen oder Schreiben) wird als Lesen und Schreiben auf die virtuelle Speichervariable `mem` angesehen. Die Trennung von Schreib- und Lesezugriffen wird in der Abbildung zu einem Ausdruck zusammengefasst. So stellt beispielsweise der Ausdruck `mem2←1` im Kontrollflussblock kb_1 den lesenden Speicherzugriff auf die virtuelle Speichervariable `mem1` und gleichzeitig den schreibenden auf `mem2` dar. Beide virtuellen Zugriffe sind dem Feldzugriff `a[1]` zugeordnet. In der Abbildung sind Anweisungen mit enthaltenen Speicherzugriffen noch einmal in der SSA-Notation mit virtuellen Speichervariablen in Klammern geschrieben. Die Zugriffe auf die Variable `mem` werden als skalare Variablen angesehen. Auch hier werden Φ-Anweisungen an Stellen mit zu vereinenden Variablen-Definitionen eingefügt.

Durch die Erzeugung von eindeutigen Variablen-Namen ergeben sich für das Compilieren Vorteile sowohl für die Erstellung von SW wie auch von HW. So werden die Lebenszeiten von Variablen verkürzt. Weiterhin wird jede Variable nur an einer Stelle im Programm beschrieben. Das verbessert die Laufzeit von Optimierungen und erlaubt eine bessere Datenflussanalyse eines Programms. Der Datenfluss ist implizit in die SSA-Form eingebunden und mit dem Kontrollfluss im CFG verbunden.

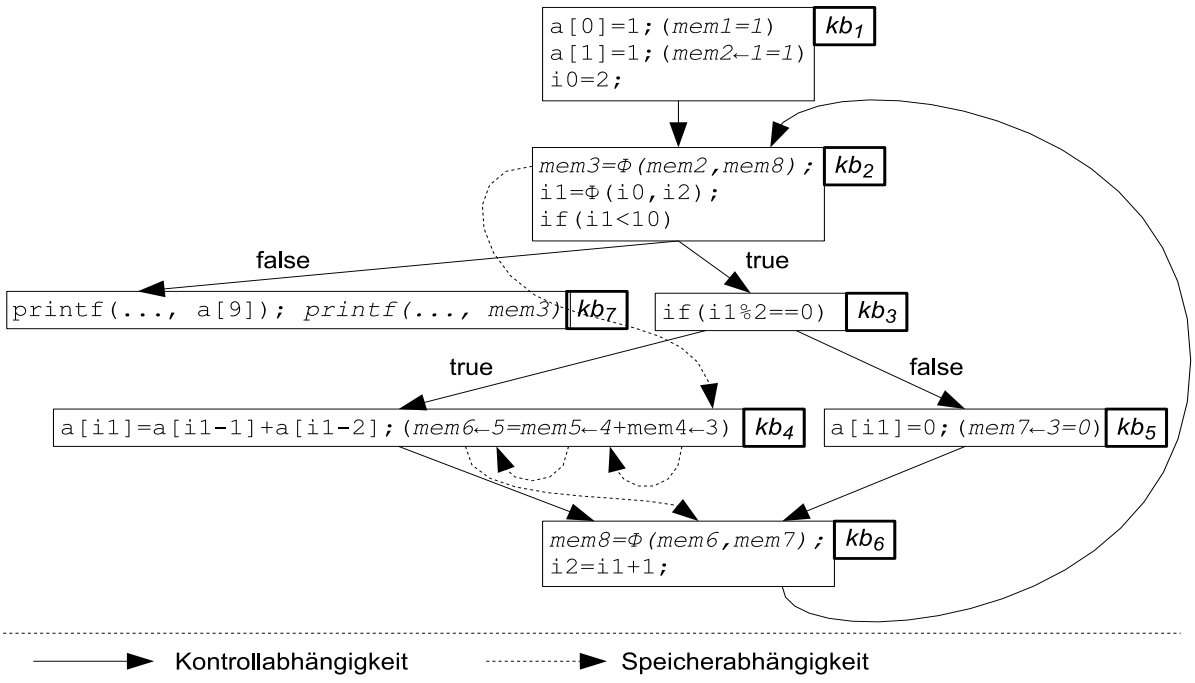


Bild 4.4 SSA-Form von Beispiel 4.3

4.2.3 Dataflow-Controlled SSA-Form

Während der Implementierung von COMRADE wurden Versuche mit einer Ausprägung der SSA-Form gemacht. Ziel dieser neuen Form war die verstärkte Einbindung von Datenfluss in die Repräsentation. Die Dataflow-Controlled SSA-Form (DFCSSA-Form) [KKGR03] hat Vorteile gegenüber einer normalen SSA-Form wegen der Reduzierung von verwendeten Φ-Anweisungen.

Der Unterschied der DFCSSA-Form zur SSA-Form ist die Platzierung der Φ-Anweisungen. Diese werden nicht wie in der normalen SSA-Form in dem Kontrollflussblock platziert, in dem sich von unterschiedlichen Vorgängern im CFG unterschiedliche Zuweisungen einer Ursprungsvariablen treffen würden. Die Φ-Anweisungen werden vielmehr direkt vor die Verwendung einer Ursprungsvariablen platziert, wenn sich dort mehrere Definitionen treffen.

```
int main() {
  int i = 0;
  int a = 0;
  do {
    if( i < 15 ) {
      if( i < 10 ) a = 1;
    } else         a = 2;
    c = a + 1;
    i = i + 1;
  } while( i < 100 );
}
```

Beispiel 4.5 while-Schleife

Betrachtet man nun die aus Beispiel 4.5 entstehenden CFGs in SSA- und DFCSSA-Form, so ist die teilweise Optimierung der Darstellung in der DFCSSA-Form (Bild 4.6b) erkennbar. Die Φ-Anweisung für `a4` wird gleich mit vier Argumenten aufgebaut. In der normalen SSA-Form sind hierzu drei Φ-Anweisungen notwendig. Der Vorteil dieser Darstellung für die HW-Erzeugung liegt darin, dass keine Hierarchien von Multiplexern aufgebaut werden müssen (`a4`, `a5`). Man kann für die Multiplexer mit mehr Eingängen gleich eine in der Laufzeit günstige Implementierung wählen. Statt einer Hierarchie von Multiplexern mit zwei Eingängen wird dann nur ein, gegenüber der Hierarchie schnellerer, Multiplexer mit mehr Eingängen verwendet.

Die Eigenschaft der Erzeugung von Φ-Anweisungen direkt vor der Verwendung einer Variablen kann aber auch zur Vergrößerung der Anzahl von Φ-Anweisungen beitragen. Im Beispiel werden für die Variable `i` in der DFCSSA-Form drei Φ-Anweisungen erzeugt, wohingegen die normale SSA-Form nur eine Φ-Anweisung benötigt. Dass dieser Umstand kein Problem ist, zeigt die durch das Zusammenfassen Φ-Anweisungen mit gleichen Argumenten entstandene optimierte Version (Bild 4.6c), in der nur noch zwei Φ-Anweisungen existieren.

Alle anderen wurden durch Zuweisungen ersetzt (z.B. `i3=i1`). Diese Zuweisungen erzeugen im resultierenden DFG Verdrahtung und beeinflussen die Größe des DFG nicht oder nur sehr geringfügig. In den Ergebnissen am Ende des Kapitels wird der Einfluss der SSA-Form auf die Größe des erzeugten DFG an Beispielen gezeigt.

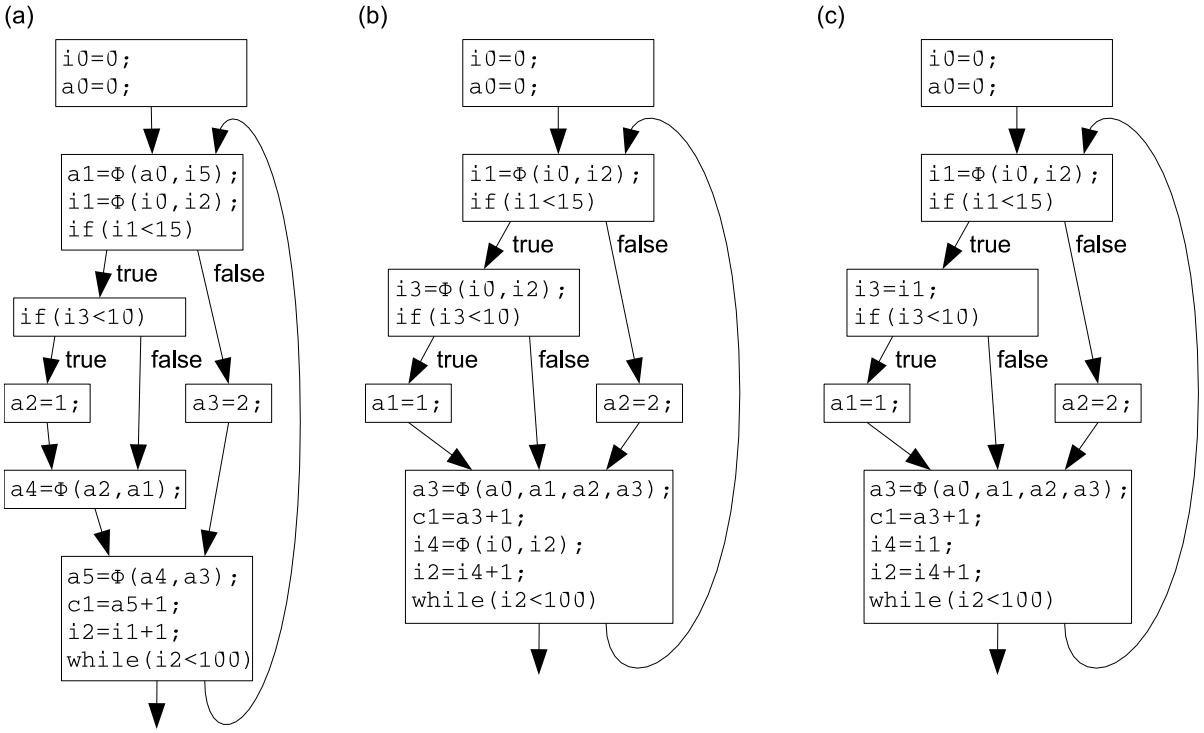


Bild 4.6 (a) SSA-Form, (b) DFCSSA-Form und (c) optimierte DFCSSA-Form von Beispiel 4.5

Trotz der Vorteile für die Größe des DFG hat die DFCSSA-Form auch einige Nachteile. Der erste Nachteil liegt in der komplizierten Handhabung von Kontrollfluss in der DFCSSA-Form. Während in der normalen SSA-Form die Φ-Anweisungen dort platziert sind, an denen abhängig vom Kontrollfluss mehrere Definitionen einer Variablen vereinigt werden müssen, befinden sich die Φ-Anweisungen in der DFCSSA-Form direkt vor den Anweisungen, welche die definierte Variable verwenden. Aufgrund dieses Umstands ist es schwierig, die Bedingungen für die Auswahl des richtigen Eingangs zu ermitteln.

Ein zweiter Nachteil ergibt sich aus Problemen im Datenfluss durch das Zusammenfassen von Φ-Anweisungen in der optimierten DFCSSA-Form. In Fällen, in denen die normale SSA-Form mehrere Φ-Anweisungen verwendet (im Beispiel `a1` und `a5`), muss die Φ-Anweisung in der DFCSSA-Form zweimal pro Schleifendurchlauf bearbeitet werden. Der Umstand rührt daher, dass die in den Φ-Anweisungen zusammengeführten Datenflüsse zwar auf gleichen Variablen beruhen, diese aber immer unter unterschiedlichen Kontrollflussbedingungen zugewiesen werden. Dies erfordert in der HW-Realisierung Register und macht die Steuerung durch den Kontroller komplizierter. Für die erste Realisierung der Datenpfade

wurde zur Erleichterung der Erzeugung des Kontrollers die im Abschnitt 4.2.2 beschriebene SSA-Form verwendet. Eine Verwendung der DFCSSA-Form in einer zukünftigen Implementierung ist aber denkbar.

4.2.4 Konvertierung der SSA-Form in einen DFG

Um die SSA-Form in einen DFG konvertieren zu können, müssen die Anweisungen in den HW-Kandidaten erst entsprechend vorbereitet werden. Ein Schritt dazu ist die Zuordnung der in den Anweisungen auftretenden Operationen zu HW-Modulen, welche auf dem Ziel-RL implementiert werden können. Dies geschieht durch FLAME-Zugriffe (Abschnitt 3.1.2). Als Vorbereitung darauf müssen Feldzugriffe in Pointer-Arithmetik konvertiert werden, da die Beschreibung von HW-Modulen für Feldzugriffe für FLAME (noch) nicht existiert. Die HW-Module stellen also im DFG die Knoten dar.

Die Erzeugung der Verdrahtung zwischen den HW-Modulen wird durch die besonderen Eigenschaften der SSA-Form gut unterstützt. So wird der Aufbau von Verdrahtungen zwischen der einzelnen HW-Modulen durch die implizite Verfügbarkeit von Datenabhängigkeiten in der SSA-Form erleichtert: Die Stellen einer Zuweisung einer Variablen werden mit den Stellen der Verwendung einer Variablen verbunden.

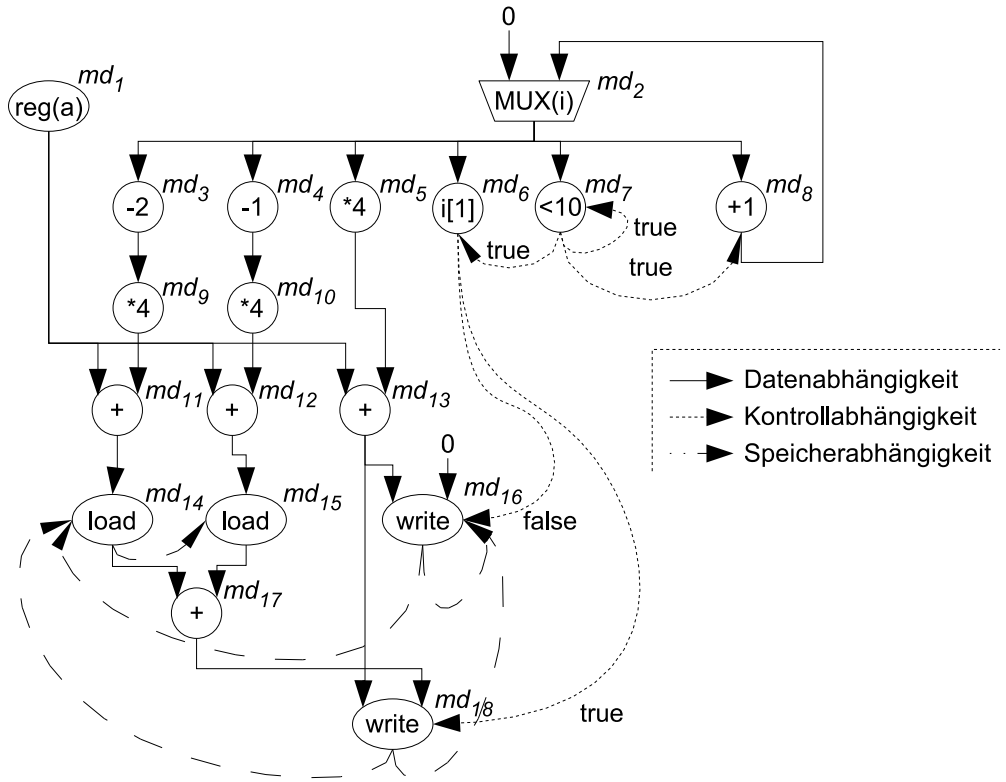


Bild 4.7 DFG der SSA-Form aus Bild 4.4

Während dieser Konvertierung werden die in der SSA-Form auftretenden Φ-Anweisungen in Multiplexer übersetzt, welche mehrere Datenflüsse miteinander mischen können. Durch diese Konvertierung ergibt sich für die Ausführung in der HW die Möglichkeit, die an einem Multiplexer endenden Datenflüsse konkurrierend spekulativ auszuführen. Sind die Daten an einem Eingang des Multiplexers frühzeitig vorhanden und die den Multiplexer steuernden Kontrollinformationen wählen diesen Eingang aus, so wird dieser vorzeitig an den Ausgang des Multiplexers geschaltet und alle anderen spekulativen Berechnungen können beendet werden. Dies kann aber nur mit einer geeigneten Unterstützung durch den Kontroller für die Ausführungssteuerung des DFG erreicht werden (Abschnitt 4.3).

Wird die Schleife in Bild 4.4 für eine HW-Ausführung vorgesehen, kann man im erzeugten DFG (Bild 4.7) die Zerlegung der Feldzugriffe auf mehrere HW-Module erkennen (`a[i]` wird zu md_3, md_9, md_{11}, md_{14}). Außerdem ist der für die Φ-Anweisung der Variablen `i` erzeugte Multiplexer (md_2) erkennbar. Dieser schaltet auf den Ausgang entweder die Konstante 0 oder das Ergebnis einer Addition (md_8) durch. Die Auswahl des richtigen Eingangs wird vom Kontroller des DFG übernommen. Dabei verwendet er die Informationen des Vergleichs-Moduls md_7. Im DFG enthaltene Kanten für die Kontroll- und Speicherabhängigkeiten werden nicht in der HW realisiert. Sie dienen vielmehr dazu, die Erzeugung des Kontrollers zu unterstützen.

4.2.5 Speicherabhängigkeiten im DFG

Eine *Speicherabhängigkeit* im DFG entsteht aus der SSA-Form dann, wenn eine Datenabhängigkeit zwischen der von einem Speicherzugriff beschriebenen virtuellen Speichervariablen (`mem`) und von einem anderen Speicherzugriff gelesenen virtuellen Speichervariablen besteht. Für die Ausführung der HW-Module am Start und Ziel einer Speicherabhängigkeit bedeutet dies, dass das HW-Modul am Ziel erst dann ausgeführt werden kann, wenn die Ausführung des HW-Moduls am Start beendet ist. Enden an einem HW-Modul mehrere Speicherabhängigkeiten, so kann dieses HW-Modul dann ausgeführt werden, wenn ein HW-Modul an einem der Ausgangspunkte der Abhängigkeiten beendet worden ist.

Dabei kann die Aktivierung von mehreren HW-Modulen an den Quellen von Speicherabhängigkeiten zu einem Ziel-HW-Modul nicht auftreten. Zum ersten entstehen mehrere eintreffende Speicherabhängigkeiten an einem HW-Modul nur aus der Existenz von Φ-Anweisungen für `mem`-Variablen. Diese bestimmen die weitergeleitete Variable kontrollflussabhängig aus einer Anzahl von Variablen. Nur eine kann dabei aktiv werden. Dazu dürfen die Speicherzugriffe natürlich nicht spekulativ ausgeführt werden. Das ist in COMRADE auch der Fall.

Im Beispiel 4.3 wird die virtuelle Speichervariable `mem7` durch den Speicherzugriff `a[i1]=0` im Kontrollflussblock kb_5 erzeugt. Diese virtuelle Variable fließt nun über die Φ-Anweisungen `mem8=Φ(mem6, mem7)` im Kontrollflussblock kb_2 und `mem3=Φ(mem2, mem8)` in Kontrollflussblock kb_2 an die Feldzugriffe `a[i1]` im Kontroll-

flussblock kb_5 und `a[i1-2]` im Kontrollflussblock kb_4. Somit entstehen im DFG Kanten für Speicherabhängigkeiten von md_{16} zu sich selbst und md_{14} (Bild 4.7).

Dabei wird eine Speicherabhängigkeit von einem HW-Modul zu einem zweiten weggelassen, wenn außerdem eine Datenabhängigkeit besteht. In diesem Fall wird die Reihenfolge der Ausführung der beiden Module allein durch die Datenabhängigkeit gewährleistet (z.B. müssen in `a[i+1]=a[i]` die Speicherzugriffe wegen des Datenflusses sowieso hintereinander ausgeführt werden). Eine zusätzliche Speicherabhängigkeit, welche auch nur die Reihenfolge der Ausführung bestimmt, ist somit nicht nötig. Dies ist beispielsweise für die Zugriffe `a[i1-1]` (md_{15}) und `a[i1]` (md_{18}) im Kontrollflussblock kb_4 der Fall. Normalerweise wäre hier eine Speicherabhängigkeit in den DFG einzufügen. Da aber von md_{15} über md_{17} nach md_{18} auch eine Datenabhängigkeit besteht, kann die Speicherabhängigkeit vernachlässigt werden.

Bei den Speicherabhängigkeiten werden keine Unterschiede zwischen Feldzugriffen auf unterschiedliche Felder sowie Speicherzugriffen auf Grund der Dereferenzierung von Pointern gemacht. Dies stellt sicher, dass alle Speicherzugriffe auch dann in korrekter Reihenfolge ausgeführt werden, wenn sich die durch unterschiedliche Variablen referenzierten Speicherbereiche überschneiden. Durch geeignete Analysen der Speicherbereiche (Abschnitt 5.1) können die Abhängigkeiten zwischen den Speicherzugriffen aber wieder entfernt werden. Dadurch ergibt sich die Möglichkeit von parallelen Zugriffen auf den Speicher, wenn die Speicherarchitektur des ACS es erlaubt. Weiterhin sind dann auch spekulative Ladezugriffe auf den Speicher möglich.

4.2.6 Kontrollabhängigkeiten im DFG

Grundlegend für die Erzeugung des Kontrollers im Datenpfad sind die *Kontrollabhängigkeiten* zwischen den einzelnen HW-Modulen im DFG. Diese werden vom Kontroller für die Erzeugung der Steuerung der Multiplexer und die Ausführung von Speicherzugriffen verwendet. Der Start-Knoten einer Kontrollabhängigkeit ist immer ein HW-Modul, welches eine Kontrollinformation erzeugt. Die in den DFG eingefügten Kontrollabhängigkeiten haben je nach Ziel-Knoten unterschiedliche Bedeutung.

Ist der Ziel-Knoten ein HW-Modul, welches selbst keine Kontrollinformation erzeugt, so ist der Ausgang mit einem Multiplexer verbunden ($md_7 \rightarrow md_8$). Während der Ausführung der Datenpfade steuert der Kontroller die Module nun so, dass das Ergebnis dieses HW-Moduls nur an den folgenden Multiplexer weitergegeben wird (das Register am Ausgang schaltet und der Multiplexer wählt den Ausgang als Eingang), wenn die Kontrollflussinformation es erlaubt.

Ist der Ziel-Knoten ein HW-Modul, welches selbst eine Kontrollinformation erzeugt, so entstehen hierdurch Kaskaden von aufeinander aufbauenden Kontrollinformationen ($md_7 \rightarrow md_6$, $md_7 \rightarrow md_7$). Die Kontrollinformation des Ziel-Knotens kann hierbei durch den Kontroller nur dann ausgewertet werden, wenn die Kontrollinformation des Start-Knotens wahr war. Ansonsten ist die Kontrollinformation des Ziel-Knotens mit falsch gleichzusetzen.

Ist der Ziel-Knoten ein Speicherzugriff, so wird dieser erst ausgeführt, wenn die Kontrollflussinformation am Start-Knoten wahr ist. Somit können Schreibzugriffe vermieden werden, die beispielsweise falsche Werte in den Speicher schreiben. Das Lesen von Speicherstellen, die noch einen alten Wert enthalten, wird durch das Einbinden der Speicherabhängigkeiten in den Kontroller vermieden.

In verschiedenen anderen Projekten wird die SSA-Form verwendet, um aus den darin enthaltenen Kontrollflussinformation HW-Implementierungen zu erzeugen [TuPa95, CaEA99, BuGo02]. In COMRADE wird versucht, alle Daten soweit spekulativ zu berechnen wie es die Datenabhängigkeiten zulassen. Die Schaltbedingungen für Multiplexer werden auch über zurückgerichtete Kanten im CFG transportierte Kontrollinformationen ermittelt, so dass sich hier auch Möglichkeiten von Pipelining über Schleifengrenzen hinweg ergeben.

Das Mitführen von Kontrollinformationen mit den Φ-Anweisungen [BuGo02] wie auch Versuche zur Erzeugung der Kontrollinformationen auch über Schleifengrenzen hinweg zeigten, dass Verfahren basierend auf Kontrollflussblöcken eines CFG nicht für die Behandlung von Φ-Anweisungen geeignet sind. Der Grund liegt in der kantenbasierten Auswahl der Argumente in den Φ-Anweisungen. Ein Argument der Φ-Anweisung ist dann das Resultat dieser, wenn beim Aufruf der Programmfluss über die mit dem Argument verbundenen Kante verläuft. Auf Grund dieses Umstands wurde eine veränderte Darstellung des CFG, der **Kanten-Kontrollflussgraph** (ECFG), eingeführt. Dieser ist ein gerichteter Graph und wird aus dem CFG erzeugt. Für jede Kante im CFG enthält der ECFG einen Knoten. Somit kann also jeder Knoten im ECFG auch durch Start-CFG-Knoten und Ziel-CFG-Knoten identifiziert werden. Eine Kante im ECFG zeigt an, dass der Ziel-CFG-Knoten eines ECFG-Knoten mit dem Start-CFG-Knoten eines anderen ECFG-Knoten übereinstimmt.

Aus dem ECFG sind ähnlich wie beim CFG nun die Beziehungen der einzelnen Kanten des CFG durch verschiedene Knotenrelationen erkennbar (Bild 4.8). So kann beispielsweise durch eine Dominatorrelation im ECFG herausgefunden werden, dass die Kante $kb_2 \rightarrow kb_3$ ausgeführt werden muss, bevor die Kante $kb_6 \rightarrow kb_2$ ausgeführt werden kann. Diese wird aber nur betreten, wenn die Bedingung in Kontrollflussblock zwei (`i1<10`) wahr ist. Somit ist die Aktivierungsbedingung des Eingangs von md_8 nach md_2 durch md_7==`true` in Bild 4.7 ermittelbar.

Diese Dominatorrelationen zwischen den Knoten des ECFG werden somit auch benutzt, um die Kontrollabhängigkeiten im DFG zu ermitteln. Dabei wird zwischen Φ-Anweisungen und anderen HW-Modulen (Ein-/Ausgaberegister, Speicherzugriffe, HW-Module für Kontrollinformationen) unterschieden.

Für Φ-Anweisungen ist das Erkennen der Aktivierung eines bestimmten Eingangs wichtig. Deshalb werden hier auch die Kanten von den Vorgängern der Kontrollflussblöcke betrachtet, in denen sich die Φ-Anweisungen befinden. Jede dieser Kanten aktiviert ein Argument der Φ-Anweisungen. Um die Bedingung zum Betreten dieser Kante herauszufinden, wird für sie im Dominatorbaum des ECFG der nächstgelegene Dominator gesucht, dessen zugeordnete Kante durch eine Bedingung betreten wird.

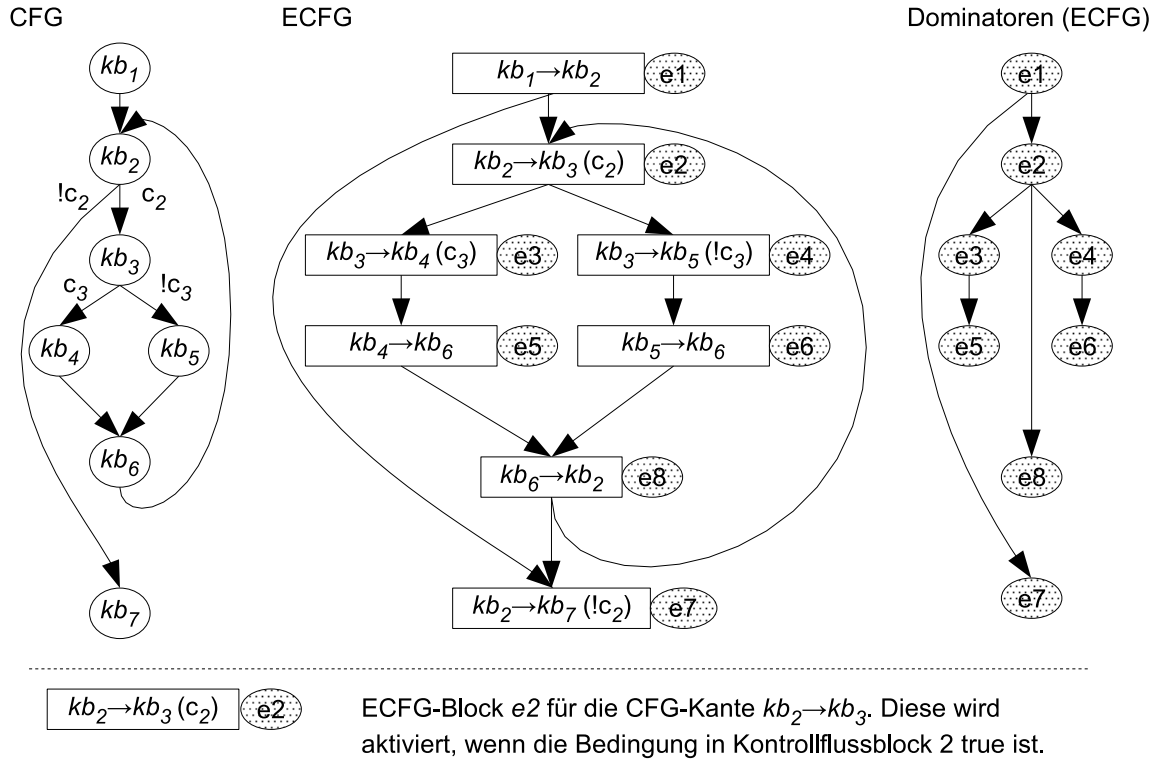


Bild 4.8 CFG, ECFG und Dominatorbaum des ECFG der SSA-Form aus Bild 4.4

Nehmen wir beispielsweise an, dass sich eine Φ-Anweisung in Kontrollflussblock kb_2 befindet. Ein Argument wird beispielsweise in kb_1 und eines in kb_6 erzeugt. Das Argument von kb_1wird über die Kante *e1* transportiert, das Argument von kb_6 über Kante *e8*. Die Kante *e1* hat im Dominatorbaum des ECFG keinen Dominator mehr, die Kante wird zu Beginn der Ausführung aktiv. Die Kante *e8* hat im Dominatorbaum den Vorgänger *e2*, welcher durch eine Bedingung aktiviert wird (c_2). Somit kann die Φ-Anweisung nur das Ergebnis der Berechnung in kb_6 übernehmen, wenn die Bedingung c_2 berechnet und erfüllt ist.

Für die anderen HW-Module ist es wichtig, in welchen Kontrollflussblöcken die Operationen enthalten sind, aus denen die HW-Module entstanden sind. Für die Suche nach den Dominatoren des ECFG werden alle Kanten betrachtet, die in diesem Kontrollflussblock enden.

Durch diese Vorgehensweise entstehen für Kontrollinformation Hierarchien wie in [KoWo02] beschrieben. Teilweise können diese aber auch zusammengefasst werden und somit die Ausführung beschleunigen. Das kann beispielsweise für HW-Module vorgenommen werden, welche sich im selben Schleifenkörper befinden [BuGo02].

Eine besondere Situation kann auftreten, wenn das Ergebnis eines HW-Moduls von mindestens einem Multiplexer und auch anderen HW-Modulen benötigt wird (siehe HW-Modul md_1 in Bild 4.12). Hierbei wird die Weitergabe des Ausgangswerts des HW-Moduls an den oder die Multiplexer durch eine Kontrollabhängigkeit reguliert. Die Übergabe an ein normales HW-Modul geschieht aber ohne Einschränkung. In diesem Fall wird der DFG so verändert, dass das HW-Modul kein Register mehr am Ende des kombinatorischen Pfads besitzt. Für

dieses eine Register werden dann der Anzahl von ausgehenden Kanten im DFG entsprechend Register eingefügt. Nur noch die Register, welche Daten für einen Multiplexer berechnen, erhalten dann eine Kontrollabhängigkeit.

Mit Hilfe der Kontroll- und Speicherabhängigkeiten kann nun der Kontroller erzeugt werden.

4.3 Erzeugen des Kontrollers

Der für die Datenpfade verwendete Kontroller ist eine Erweiterung von Petri-Netzen [Pete81]. Die folgenden Abschnitte beschreiben den Aufbau des Kontrollers sowie dessen Arbeitsweise. Auch in anderen Projekten [BuGo02, Buck93] werden Petri-Netze zu Grunde gelegt. Die Grundidee bei allem ist, dass ein HW-Modul nur dann ausgeführt werden kann, wenn die Ergebnisse der Vorgänger-Module im DFG vorliegen. Das unterscheidet sie von Modellen, welche FIFOs zwischen den HW-Modulen vorsehen, um so Daten zwischenspeichern zu können [Kahn74].

```
int y = 100;
do {
  if( y == 90 ) y=y/3;
  else          y=y-1;
} while( y > 20 );
```

Beispiel 4.9 do-while-Schleife

Ein Modell ohne spekulative Ausführung verwendet für Berechnungen, die unter verschiedenen Kontrollflussbedingungen ausgeführt werden, *switch*- und *select*-Module [Buck93]. Ein Zweig der Berechnungen wird hier nur dann durch einen switch-Modul begonnen, wenn die Kontrollinformationen dies zulassen (Bild 4.10a). An select-Modulen werden die Berechnungen wieder zusammengefasst. In diesem Fall muss man immer auf die Berechnung von Kontrollinformationen warten, bis die Berechnungen begonnen werden können. Außerdem werden berechnete Kontrollinformationen oft an mehreren Stellen verwendet, was die Verdrahtungskapazitäten auf dem Ziel-RL beansprucht.

Eine Erweiterung der auf switch-select-Modulen bestehenden Modelle lässt die spekulative Berechnung in den durch Kontrollinformationen getrennten DFG-Bereichen zu. So könnten beispielsweise `/3` und `-1` spekulativ parallel ausgeführt werden. Diese spekulative Ausführung kann aber zu Problemen führen, wenn die Berechnungen auf unterschiedlichen Wegen durch den DFG in den Laufzeiten stark unterschiedlich sind (`/3` benötigt mehr Zeit als `-1`). Ohne besondere Vorkehrungen bei der Steuerung des Ablaufs muss an der Vereinigung durch einen Multiplexer immer auf die Fertigstellung der längsten Berechnung gewartet werden.

Eine dieser Vorkehrungen besteht darin, jeden Eingang eines Multiplexers mit einem Zähler zu versehen. Dieser kann die Anzahl von nicht verwendeten Ergebnissen zählen. Erreichen die Ergebnisse den Multiplexer, werden sie verworfen. Außer der für die Zähler nötigen

HW-Ressourcen kann es dazu kommen, dass es in diesen Zählern einen Überlauf gibt und somit das durch die HW berechnete Ergebnis falsch ist.

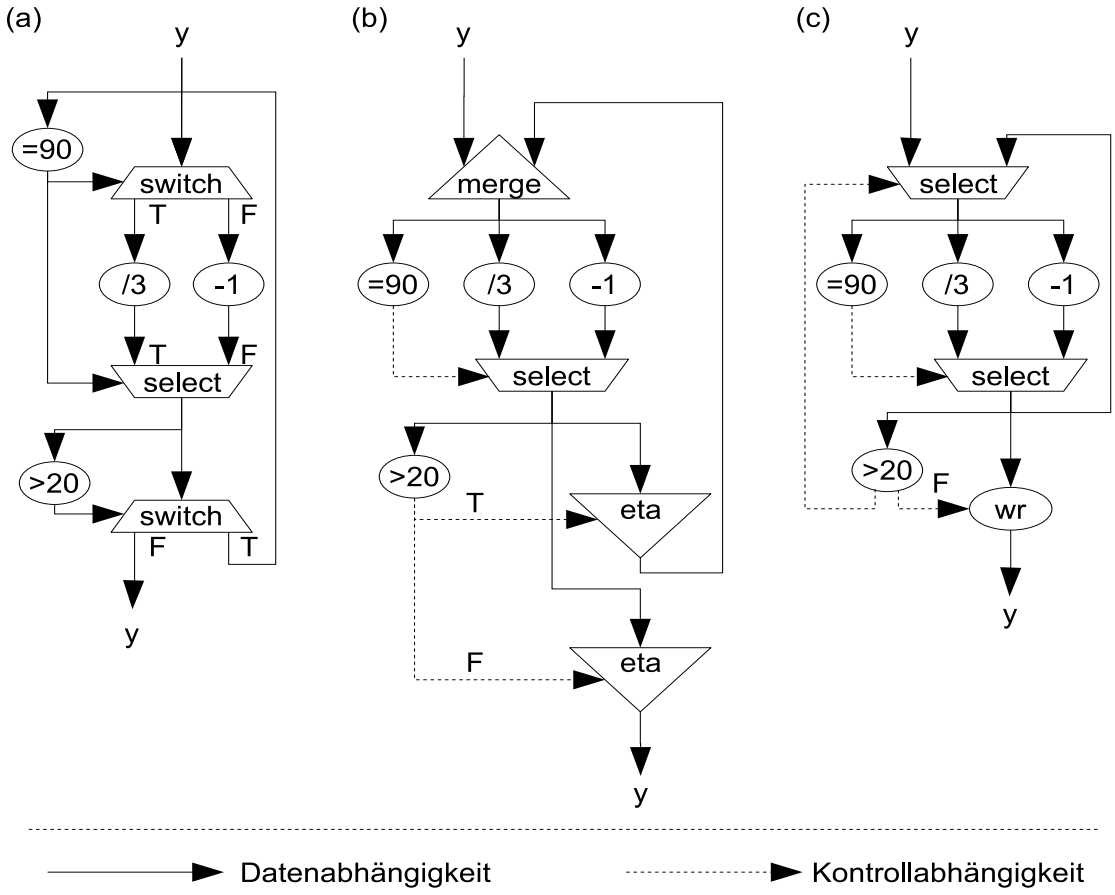


Bild 4.10 Beispiel 4.9 als (a) Tokenfluss-Modell, (b) Pegasus-Darstellung, (c) COMRADE-DFG

Eine andere Lösung ist das Verbinden jedes HW-Moduls mit einem so genannten Sequenz-Token, in dem die aktuelle Schleifeniteration gespeichert ist [StLu03]. Durch dieses Token ist es möglich, auf unterschiedlich schnellen Pipelines errechnete Ergebnisse aufgrund der Information im Token zu ordnen und nicht benötigte Daten zu verwerfen. Neben dem Platzverbrauch für die Logik zum Auswerten der Sequenz-Token benötigen die Token selbst Platz auf der Ziel-RL. Des Weiteren werden auch hier noch switch-Module verwendet, um die Berechnung in den unterschiedlichen Kontrollflusszweigen mit unterschiedlichen Bedingungen zu verknüpfen. Spekulative Ausführung ist somit nur durch hohen Aufwand realisierbar.

Eine Erweiterung zu dem beschriebenen Datenflussmodell stellt das Entfernen der switch-Module in Pegasus dar [BuGo02](Bild 4.10b). Aus spekulativ berechneten Werten wird an einem Multiplexer der richtige ausgewählt. Die HW-Module haben bei den Berechnungen immer eine Verbindung zu den Vorgängern und Nachfolgern im DFG und wissen

selbstständig, wann sie mit der Berechnung anfangen können oder ein Ergebnis von einem Nachfolger übernommen wurde. Somit unterstützt dieses Modell auch Pipelining.

Für das Zirkulieren von Daten im DFG, welche von einer Schleifeniteration zur nächsten übergeben werden, werden in Pegasus so genannte *eta*- und *merge*-Knoten verwendet. Die eta-Knoten geben den Wert einer Variablen erst in die Berechnung einer nächsten Schleifeniteration, wenn die Bedingung dazu gegeben ist. Der merge-Knoten mischt zwei Datenflüsse im Unterschied zum select-Knoten ohne Kontrollflussinformationen miteinander. Auch in Pegasus treten Probleme bei der spekulativen Ausführung von Modulpfaden mit stark unterschiedlichen Laufzeiten auf, da an den Multiplexern immer auf die Beendigung des längsten Modulpfads gewartet werden muss. In dem in Bild 4.11 werden verschiedene Möglichkeiten der Behandlung von spekulativer Berechnung aufgezeigt. Dargestellt ist ein Ausschnitt aus einer Schleife im DFG. Jedes HW-Modul benötigt hier einen Takt zur Berechnung außer das mit der Division durch drei (drei Takte). Dabei können zu jedem Takt Daten am Eingang zu den beiden ersten HW-Modulen erwartet werden. Außerdem wird durch das select-Modul immer das Ergebnis der schnellsten Berechnung gewählt. Wird nun immer auf die Beendigung der Berechnung in beiden spekulativen Zweigen gewartet (Bild 4.11a), dann kann während der gezeigten fünf Schritte nur ein einziges Datum berechnet werden.

Als Lösung könnte man sich die Möglichkeit überlegen, ein select-Modul zu schalten, wenn Kontroll- und Datenflussinformationen vorhanden sind. Länger laufende Berechnungen würde man dann einfach laufen lassen und beim Erreichen des select-Moduls eliminieren. Dabei kann ein Zähler festhalten, wie viele Daten nicht mehr benötigt werden (Bild 4.11b). Damit können nun schon mehr Daten während eines Zeitabschnitts berechnet werden. Ein Problem tritt hierbei in Schleifen auf. So kann es vorkommen, dass sich die Berechnungen in den längsten Modulpfaden stauen, wenn immer der kürzeste Modulpfad gewählt wird. Der längste würde aber bei einem statischen Ausführungsmodell in jedem Fall auch ausgeführt. Da in unserem Beispiel mit einem Datum in jedem Takt gerechnet werden kann, staut sich die Berechnung das erste Mal in Schritt zwei, da das Modul mit der Division noch rechnet. Diese kann also das Ergebnis des Vorgängers nicht übernehmen und somit auch der kurze Zweig nicht. Erst in Schritt vier wird das Ergebnis der Division weitergegeben und damit auch ein neuer Eingabewert übernommen. Erst jetzt kann auch der kurze Zweig weiterrechnen. Dessen Ergebnis kann nun in Schritt fünf an das select-Modul weitergegeben werden.

Eine Abhilfe für die oben genannten Nachteile bringt das in COMRADE verwendete, erweiterte Modell des Petri-Netzes (Bild 4.10c). Neben den Token, welche eine Vorwärtsbewegung im DFG beschreiben (*Up-Token*), unterstützt der Kontroller in COMRADE auch Token, welche sich entgegen dem Datenfluss bewegen können (*Down-Token*). Aufeinander treffende Up- und Down-Token löschen sich gegenseitig aus. Damit können die negativen Effekte von unterschiedlich langen Modulpfaden während der spekulativen Berechnung vermindert werden.

Im Beispiel (Bild 4.11c) weiß der Kontroller in Schritt eins, dass der kurze Weg gewählt wird. Damit wird für den langen Weg ein Down-Token erzeugt, welches sich jetzt entgegengesetzt der Datenflussrichtung ausbreiten könnte. Im Beispiel löscht das Down-Token die

Berechnung der Division schon in Schritt zwei. Somit kann in Schritt drei schon ein neues Datum spekulativ berechnet werden. Das Ergebnis der Berechnung des zweiten Datums liegt somit schon in Schritt vier am select-Modul an. Ein ganzer Takt konnte hier in nur fünf Takten eingespart werden.

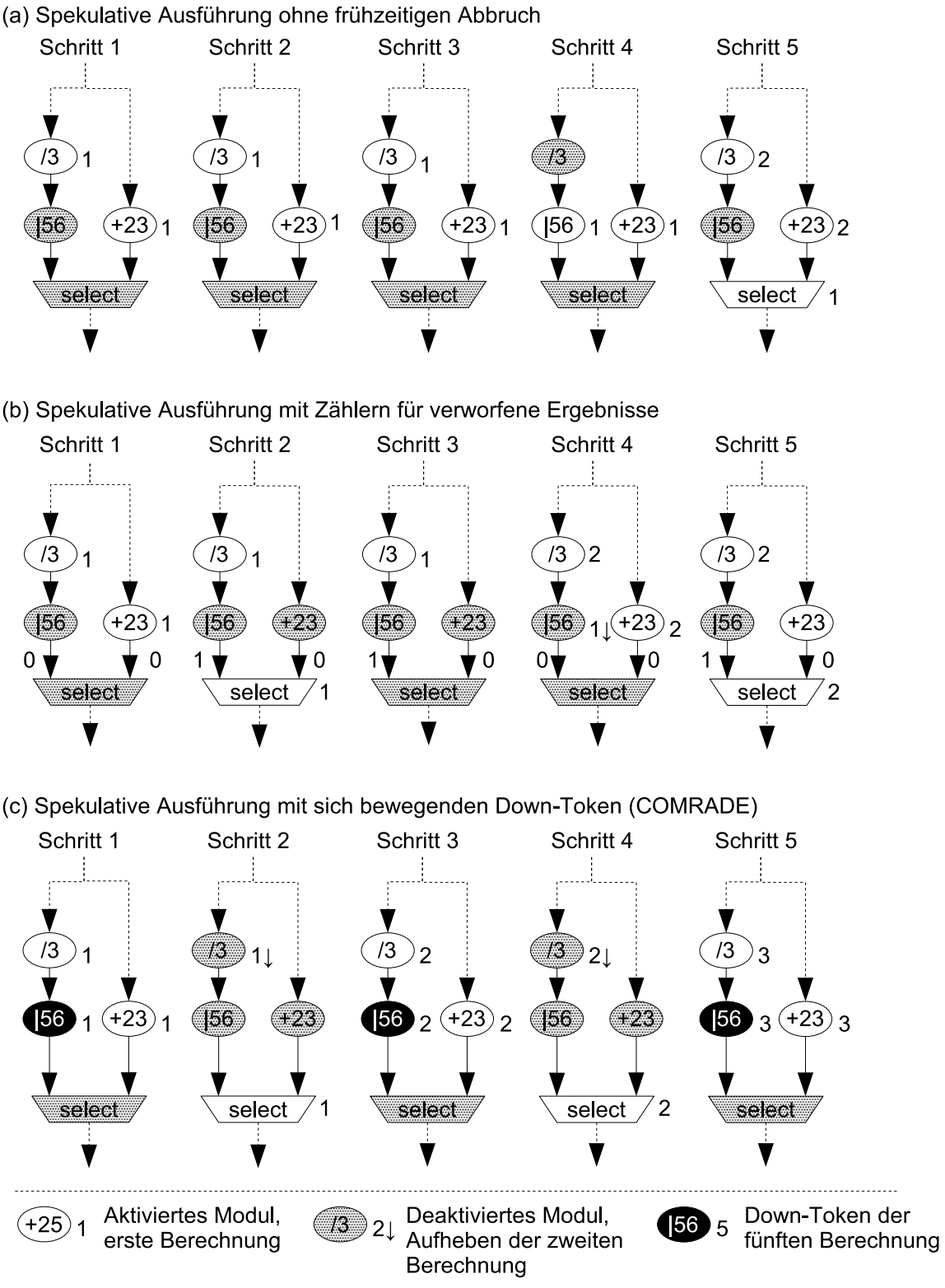


Bild 4.11 Spekulative Ausführungsmöglichkeiten

Außerdem bestehen in COMRADE nicht mehr die Unterschiede zwischen merge- und eta-Knoten. Diese werden wie auch alle anderen sich vereinenden Datenflüsse durch einen Multiplexer ersetzt (Bild 4.10c) und reduzieren den Aufwand für die HW-Implementierung.

4.3.1 Ausführungsmodell des Kontrollers

Die Struktur des Kontrollers in COMRADE ähnelt der des DFG. Obwohl in der Implementierung der DFG und der Kontroller getrennt voneinander verwaltet werden, kann man die Funktionsweise so betrachten, dass jedes HW-Modul eine bestimmte Anzahl von Token erzeugt und auch verbraucht. Dabei kann jedem HW-Modul in einem Takt der Ausführung des DFG nur ein Up- oder ein Down-Token zugewiesen werden. Jedem HW-Modul, welches zu Beginn der Ausführung aktiv sein muss, werden beim Start des Kontrollers Up-Token zugewiesen.

Im Kontroller kann jedes Token nur einmal pro Ausführung eines HW-Moduls erzeugt und nur einmal verbraucht werden. Werden für ein HW-Modul Up-Token erzeugt, so sind die Daten am Ausgangsregister des jeweiligen HW-Moduls gültig. Dabei sind jedem HW-Modul genau so viele Up-Tokens zugeordnet, wie auch Kanten von ihm entspringen (Daten-, Kontroll- und Speicherabhängigkeiten). Dadurch wird für alle abhängigen HW-Module je ein Up-Token bereitgestellt. Ein HW-Modul (außer Multiplexer) kann nur dann ausgeführt werden, wenn ihm von jedem Vorgänger im DFG ein Up-Token zur Verfügung steht. Beim Ausführen des HW-Moduls werden diese Up-Token verbraucht.

Ein HW-Modul kann beim Vorhandensein aller Token der Vorgänger nicht ausgeführt werden, wenn für dieses HW-Modul noch mindestens ein Up-Token nicht von einem Nachfolger verbraucht worden ist. Solange Berechnungen in Abhängigkeit von Ausgangsdaten eines HW-Moduls stattfinden, darf dieser natürlich nicht mit einem neuen Wert aktualisiert werden.

Ist ein HW-Modul von einem anderen HW-Modul kontrollabhängig, so darf es natürlich nicht ausgeführt werden, wenn nicht die richtige Bedingung zur Ausführung dieses Moduls vorliegt. In diesem Fall werden Down-Token erzeugt. Diese Token beinhalten die Information, dass der an dem aktuellen HW-Modul endende Datenfluss abgebrochen werden kann. Wenn alle Up-Token der Vorgänger-HW-Module aktiviert sind, so löschen sich diese und das Down-Token aus. Die Anzahl der durch ein HW-Modul erzeugbaren Down-Token ist gleich der Anzahl von Up-Token des HW-Moduls, damit die Auslöschung für jedes Up-Token des HW-Moduls separat geschehen kann. Alle zu einem HW-Modul gehörenden Tokens werden zu einer **Tokengruppe** zusammengefasst.

Ein Down-Token im Datenflussmodell von COMRADE kann aber nicht nur erzeugt werden und auf ankommende Up-Token warten. Um spekulative Berechnungen abbrechen zu können, müssen sich die Down-Token wie auch die Up-Token bewegen können. Die Bewegung der Down-Token findet entgegen der Datenflussrichtung im DFG statt, da die Aufgabe der Down-Token das Abbrechen von Berechnungen innerhalb des DFG ist.

Bei Kontrollabhängigkeiten im DFG haben die Down-Token eine weitere Bedeutung. Ein Down-Token pflanzt sich auch in Richtung der Kontrollabhängigkeiten fort. Ein HW-Modul, welches eine Kontrollinformation erzeugt und ein Down-Token erhält, erzeugt wiederum Down-Token. Dadurch werden von ihm kontrollabhängige HW-Module deaktiviert. Dies muss geschehen, da Datenabhängigkeiten auch unabhängig vom Kontrollfluss existieren. Somit können auch Daten berechnet werden, welche beispielsweise von mehrfach verschachtelten `if`-Anweisungen abhängig sind.

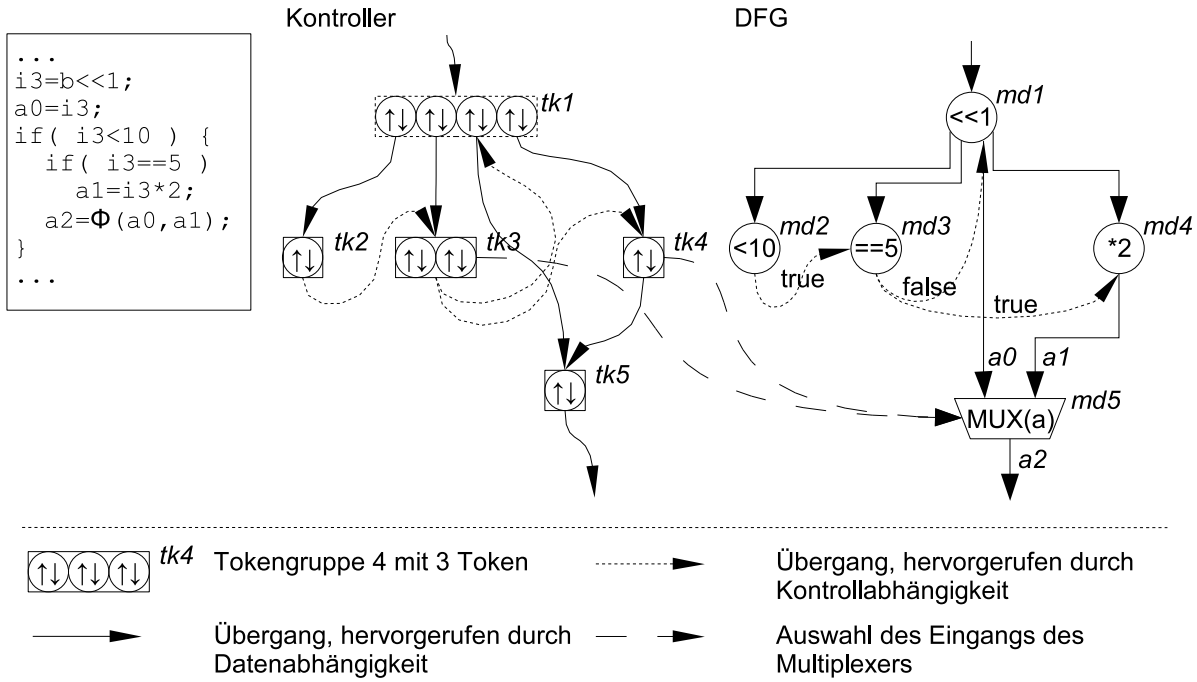


Bild 4.12 Hierarchien von HW-Modulen mit Kontrollinformationen

Das Beispiel in Bild 4.12 zeigt die Auswahl des Ergebnisses von `a2` aus `a0` und `a1` auf Grund unterschiedlicher Kontrollflussbedingungen. Nehmen wir den Fall an, dass die Bedingung `i<10` (*md2*) auf false evaluiert wurde. In diesem Fall dürfte normalerweise die Bedingung `i==5` (*md3*) nicht mehr ausgeführt werden. Da aber die Daten im DFG trotzdem berechnet werden, würde das Ergebnis von *md4* nicht verworfen werden und somit fälschlicherweise für eine folgende Berechnung im DFG existent sein (wenn *md2* auf true evaluiert wird). Damit wäre die Berechnung aber falsch.

Für diesen Fall muss also *md3* auch false evaluiert werden, wenn *md2* false ist. Damit werden von dieser Bedingung abhängige Berechnungen auch in einem solchen Fall deaktiviert. Dieser Mechanismus ist nicht im DFG-Teil des Datenpfads vorhanden. Er wird durch den Kontroller simuliert, indem ein Down-Token weitergereicht wird.

Im Bild 4.13 wurde beispielhaft ein Ausschnitt aus dem Beispiel-DFG mit dazugehörigem Kontroller dargestellt. Erkennbar ist hier die Bereitstellung der von Ausgängen im DFG gleichgesetzten Anzahl von Token. Die Tokengruppe *tk2* enthält entsprechend den sechs Ausgängen des HW-Moduls md_2 sechs mögliche Token. Diese werden alle gleichzeitig Up-aktiv, wenn alle Voraussetzungen dazu geschaffen wurden. In diesem Beispiel können sie nur

Up-aktiviert werden, wenn entweder die Up-Token von *tk1* oder *tk5* aktiv sind. Diese Up-Token sind nun getrennt voneinander in den abhängigen HW-Modulen (*tk3*, *tk4*, *tk5*, ...) verarbeitbar. Die Tokengruppe wird nur dann wiederholt betreten, wenn alle Up-Token verarbeitet wurden.

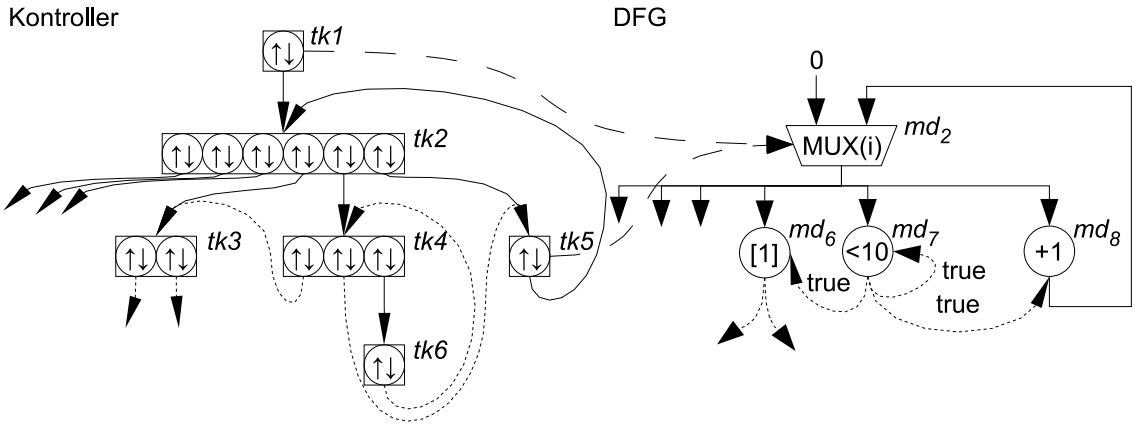


Bild 4.13 Aufbau des Kontrollers, Ausschnitt aus **Bild 4.7**

Die Tokengruppe *tk6* ist keinem HW-Modul zugeordnet. Diese wurde bei der Erzeugung des Kontrollers eingefügt, weil bei dem HW-Modul md_7 eine auf sich selbst gerichtete Abhängigkeit besteht. Da aber eine Tokengruppe für ein HW-Modul nicht aktiviert werden darf, wenn noch mindestens ein Up-Token aktiv ist, dürfte die Tokengruppe nach erstmaligem Aktivieren nicht wieder betreten werden. Das Up-Token, welches wieder für md_7 vorgesehen ist, würde die Weiterbearbeitung verhindern.

4.4 Realisierung des Kontrollers

Bei der Realisierung des Kontrollers wurden die den HW-Modulen zugeordneten einzelnen Token durch jeweils zwei Register realisiert. Dabei repräsentiert eines den Wert Up (*Up-Register*) und eines den Wert Down (*Down-Register*). Die einzelnen Register reagieren nur auf die Signale von Vorgänger- und Nachfolger-HW-Module sowie von deren Token-Registern.

Die Aktivierungsbedingung für das Up-Token eines HW-Moduls wird auch für das Schalten des Registers am Ausgang des HW-Moduls verwendet. Die Aktivierungsbedingung ist für jedes Up-Token eines HW-Moduls dieselbe (Bild 4.14a).

Eine Besonderheit bei der Ausführung des Kontrollers bilden HW-Module mit variabler Laufzeit (Bild 4.14b). Diese werden durch ein Signal gestartet (start) und signalisieren ihr Ende durch ein weiteres Signal (done). Die HW-Module werden so betrachtet, wie wenn das Up-Token an sie übergehen würde und danach wieder in den Kontroller zurückfließt. So werden diese HW-Module auch durch die Aktivierungsbedingung für die zugeordneten Up-Register gestartet. Das Up-Register wird aber nicht aktiviert. Dies geschieht erst durch das

vom HW-Modul erzeugte done-Signal. Das Up-Token ist somit wieder in den Kontroller zurückgekehrt.

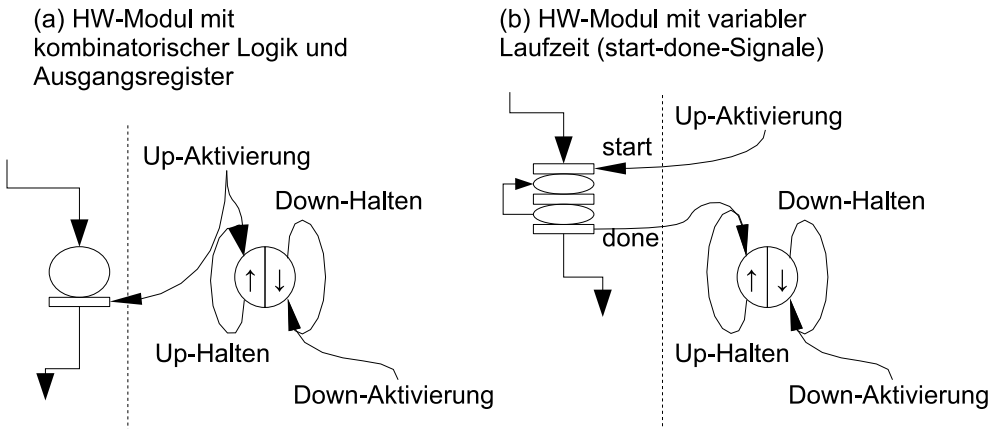


Bild 4.14 Steuerung von HW-Modulen und Token-Registern

4.4.1 Aktivierungsbedingungen der Token-Register

Für die Funktion des Kontrollers müssen die Aktivierungsbedingungen von Up- und Down-Token im Zusammenhang betrachtet werden, da sie miteinander interagieren. Da die umfassende Beschreibung der Aktivierung jedes Token-Registers hier zu langwierig wäre, werden nur die Grundlagen der Token-Register beschrieben. Eine genaue Beschreibung der für die Implementierung in COMRADE verwendeten Aktivierungsbedingungen befindet sich in Anhang A.

Alle Up-Register werden betreten und bleiben unter bestimmten Bedingungen aktiv. Alle Up-Register können nur dann betreten werden, wenn

1. es selbst nicht aktiviert ist und
2. alle Up-Register der Vorgänger-HW-Module aktiviert sind und
3. alle Kontrollflussbedingungen zur Aktivierung des HW-Moduls zutreffen.

Ohne die Existenz von Down-Token wäre mit diesen Bedingungen die Vorwärts-Bewegung des Datenflusses mittels der Up-Token beschrieben. Wenn aber ein Up-Register aktiviert werden könnte, das dazugehörige Down-Register aber schon aktiviert ist, dann muss die Berechnung verworfen werden (Bild 4.15). Die Up-Register *t1* und *t2* sowie das Down-Register *t3* werden in Schritt 1 betreten. Die verstärkt gedruckten Pfeile zeigen die Vergangenheit des Kontrollers. In diesem Zustand des Kontrollers könnten nun die Up-Register *t3* und *t4* betreten werden. Da aber *t3* schon Down-aktiviert ist, wird nur das Up-Register *t4* betreten (Schritt 2).

Nun sind die Up-Register von *t1* und *t2* nicht mehr betreten. Auch die Register der HW-Module der beiden Up-Register beinhalten jetzt Werte, welche für eine folgende Berechnung nicht mehr verwendet werden können. Deshalb sind jetzt die beiden Up-Register wieder betretbar und die Register der zugehörigen HW-Module können neue Werte übernehmen. Auf

diese Weise werden HW-Module nach einer abgeschlossenen Operation wieder für nachfolgende Berechnungen frei. Somit realisiert der Kontroller die Ausführung als eine Pipeline.

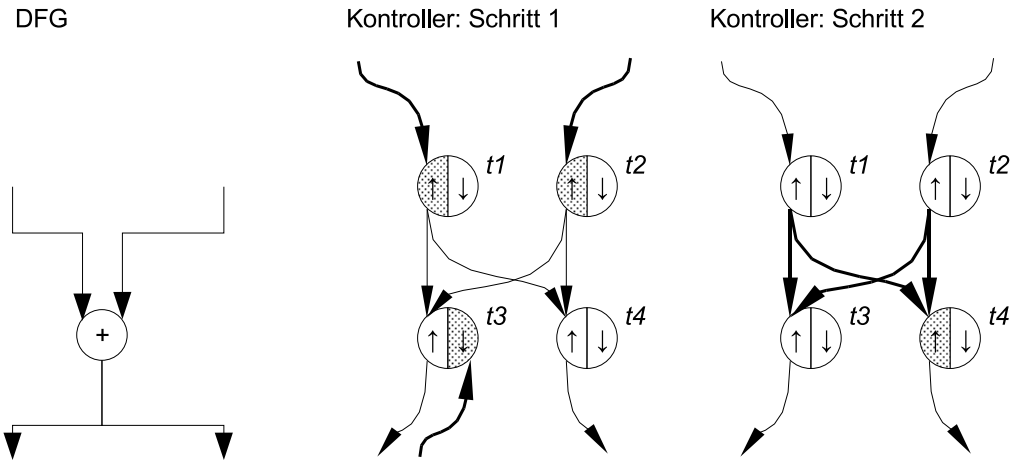


Bild 4.15 Bewegung von Up-Token in Up-Registern

Wurde ein Up-Register betreten, so muss dieses wieder verlassen werden, wenn das mit ihm verbundene Up-Token weitergereicht wird oder es mit einem Down-Token kombiniert. Das Up-Register wird also auf low gesetzt, wenn:

1. alle von ihm abhängigen Up-Register des Nachfolger-HW-Moduls aktiviert werden oder
2. alle abhängigen Down-Register des Nachfolger-HW-Moduls aktiviert sind und sich zum Up-Register bewegen könnten.

Im Beispiel kann man erkennen, dass sich beispielsweise das Up-Register von *t1* deaktiviert, weil beide von ihm abhängigen Up-Register (*t3*, *t4*) betreten werden können.

Im Gegensatz zu Up-Registern sind Down-Register auf zwei unterschiedliche Weisen aktivierbar. Ein Down-Register wird dann aktiviert, wenn

1. eine Kontrollabhängigkeit die Verwendung des Ergebnisses eines HW-Moduls behindert oder
2. die mit dem Down-Register verbundenen Down-Register von Nachfolger-HW-Modulen (aus Sicht des Datenflusses) aktiv sind und das zum Down-Register gehörende Up-Register nicht aktiv ist oder wird.

Eine Ausnahme bilden hierbei die Down-Register, welche Multiplexern zugeordnet sind (*t5* in Bild 4.16). Diese können sich nicht bewegen (2. Aktivierungsregel gilt nicht). Würde man das Bewegen von Down-Token über Multiplexer hinaus erlauben, so sind Situationen konstruierbar, in denen das Down-Token, welches von einem Multiplexer kommt ein normal erzeugtes überholt. Fehler in der HW-Ausführung wären die Folge.

Wenn ein Down-Register aktiviert wurde, muss es so lange aktiviert bleiben, bis:

1. das dazugehörige Up-Register aktiviert werden darf oder
2. sich das Down-Token entgegen der Bewegungsrichtung der Up-Token bewegt.

Im Beispiel (Bild 4.16) wird das Down-Register *t4* aktiviert, weil der Vergleich die Weitergabe des zugeordneten Ergebnisses verhindert (Schritt 1). Dazu wurden ein Up-Token (*t6*) und das durch das dazugehörige HW-Modul berechnete Ergebnis verwendet. Für den nächsten Schritt kann der Kontroller feststellen, dass kein Register in *t3* aktiviert ist. Da nur *t1* bisher evaluiert wurde und *t2* nicht, kann auch *t3* nicht im nächsten Schritt Up-aktiviert werden. Deshalb kann sich das Down-Token bewegen und das Down-Register von *t3* wird aktiviert sowie das von *t4* deaktiviert.

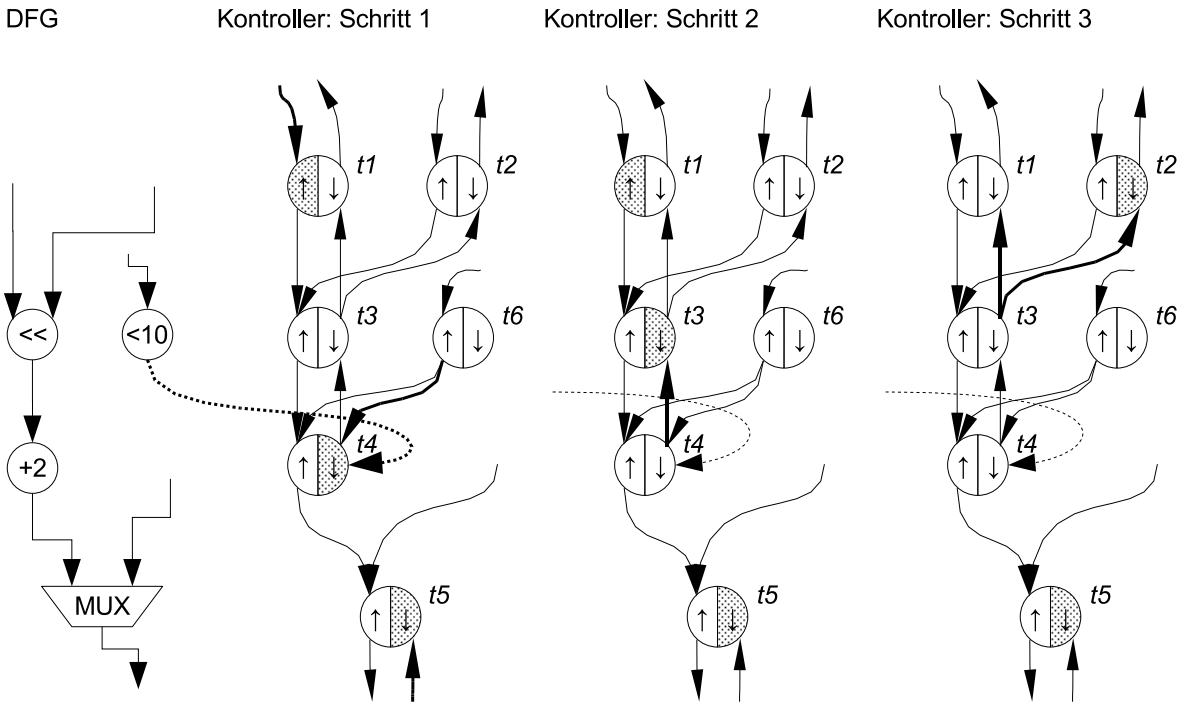


Bild 4.16 Bewegung von Down-Token in Down-Registern

Bei der weiteren Bewegung des Down-Tokens (Schritt 3) trifft es auf das aktivierte Up-Register in *t1*. Diese löschen sich aus. Nur über das Down-Register von *t2* kann es sich noch weiter bewegen.

4.4.2 Kommunikationsregister zwischen Hardware und Software

Für den Datenaustausch zwischen HW und SW müssen in den Datenpfad geeignete Mechanismen eingebaut werden, damit die Semantik von Programmen nicht verändert wird, auch wenn die HW-Ausführung für die SW-Ausführung einer SW-Subregion unterbrochen wird. Vor allem die Ausführung des Kontrollers verlangt von den Ein- und Ausgaberegistern spezielle Verhaltensweisen. Diese Register werden vom Kontroller wie HW-Module mit variabler Laufzeit aufgefasst. Alle Register haben eine eindeutige Adresse, über welche sie von der SW angesprochen werden können.

Bei der Verwendung von Eingaberegistern wird davon ausgegangen, dass ein Token-Register auf das Signalisieren der Beendigung eines Schreibzugriffs von der SW auf das Eingaberegister wartet. Das Eingaberegister reagiert auf die Kommunikation vom SW-Teil

des Programms, wenn es durch die angelegte Adresse angesprochen wird (Bild 4.17). Hat es den Wert vom SW-Teil übernommen, wird das Signal rdy auf high gesetzt. Dieses aktiviert dann die von der Eingabe abhängigen Up-Register (*t3*, *t4*).

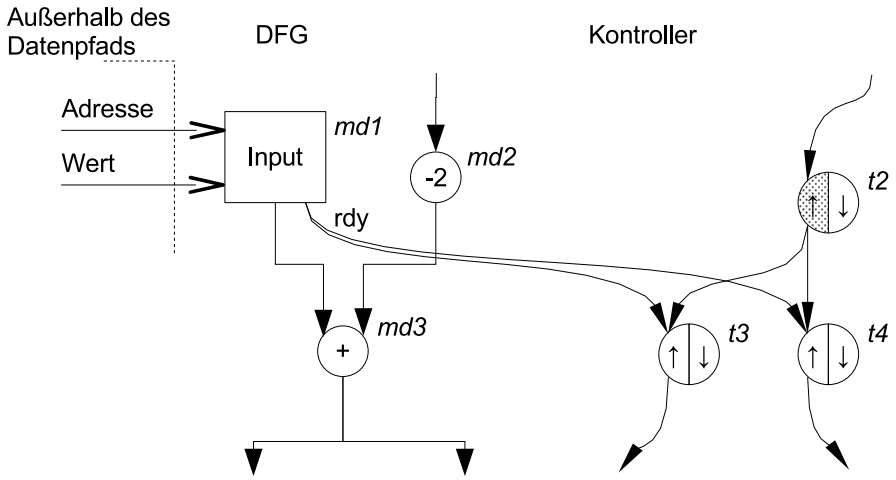


Bild 4.17 Behandlung von Eingaberegistern

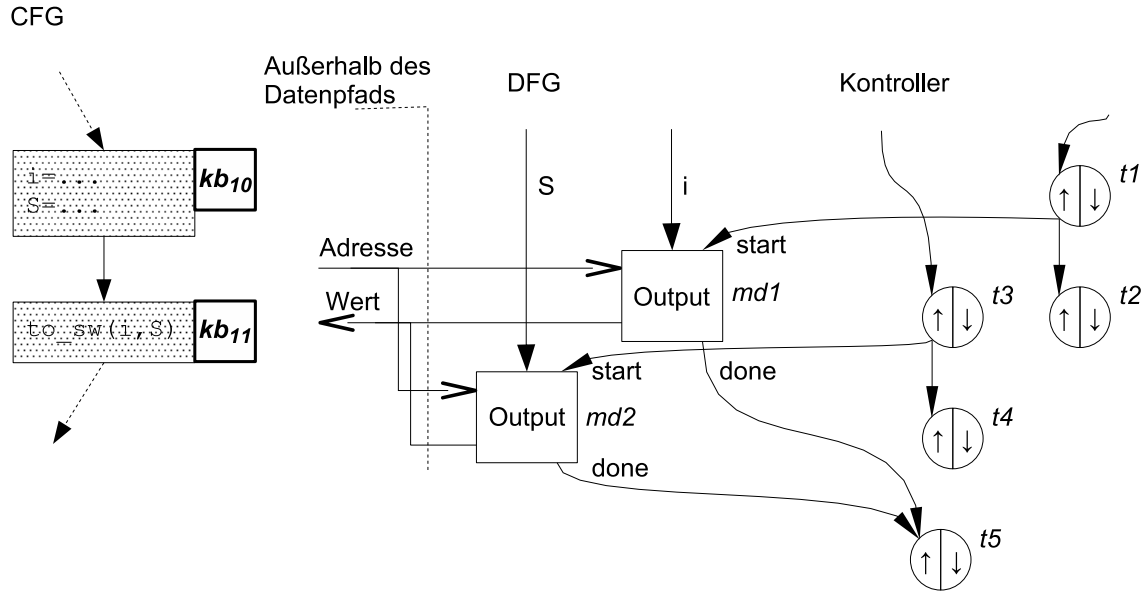


Bild 4.18 Behandlung von Ausgaberegistern

Bei der Wertübergabe mit Ausgaberegistern an die SW muss weiterhin noch beachtet werden, dass die SW-Ausführung auf einen Interrupt wartet, der sie weiterarbeiten lässt (Bild 4.18). Wenn Werte an die SW übermittelt werden sollen, wird dies den Ausgaberegistern durch ein start-Signal mitgeteilt. Nach Übergabe der Werte an den Prozessor generieren die Ausgaberegister ein done-Signal. Wenn alle done-Signale der Ausgaberegister aufgetreten sind, wird ein neues Up-Register (*t5*) betreten, welches einen Interrupt erzeugt. Mit dem Interrupt wird weiterhin ein Wert (Exit-Status) an die SW übergeben, welcher der Software signalisiert, an welcher Stelle in der Ausführung des Datenpfads dieser unterbrochen oder

beendet wurde. Die Software kann nun die von den Übergaberegistern übermittelten Werte lesen.

Dieser Wert wird der SW-Ausführung durch den Rückgabewert der Funktion `wait_for_hw()` übermittelt (Bild 4.19). Nachdem also die SW den Interrupt erhalten hat, entscheidet sich die weitere Programmbearbeitung durch den übergebenen Exit-Status (in kb_6 und kb_7). Wird durch den Exit-Status das Ende der HW-Ausführung angezeigt, dann kann die SW-Ausführung des Programms weitergehen ($kb_6 \rightarrow kb_9$).

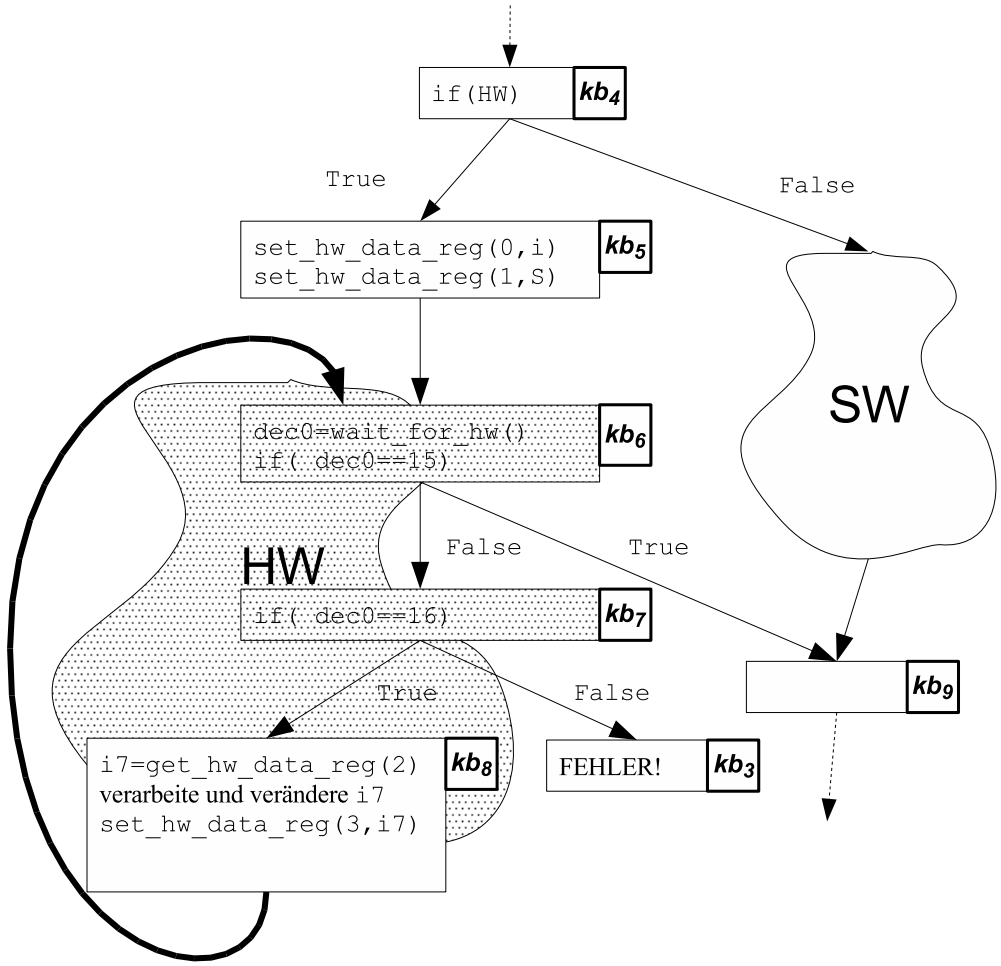


Bild 4.19 Behandlung der HW-Ausführung in SW

Soll aber nur ein Teil einer SW-Subregion ausgeführt werden, dann wird an die Stelle im Anwendungsprogramm gesprungen, welche die eingefügte SW-Subregion beinhaltet ($kb_7 \rightarrow kb_8$). Für die Ausführung der SW-Subregion werden nun alle für die Ausführung nötigen Variablen-Werte aus der unterbrochenen HW-Ausführung gelesen (`get_hw_data_reg()` in kb_8). Diese wird nun ausgeführt. Alle in der SW-Subregion veränderten Daten werden nun an die HW zurückgegeben (`set_hw_data_reg()`). Die Eingaberegister im Datenpfad warten auf die von der SW übergebenen Werte. Er kann also mit den übergebenen Daten sofort weiterrechnen. Nachdem dann alle Daten wieder in die HW-

Ausführung zurückgegeben wurden, wartet die SW wieder auf das Auslösen des Interrupts mit `wait_for_hw()`.

Sollte ein von der HW übergebener Exit-Status im generierten Programm oder im Datenpfad nicht vorgesehen sein, so wird ein Fehler erzeugt (kb_3).

4.5 Notwendige Änderungen des Quellcodes

Durch die Eigenschaften der SSA-Form sind nicht alle gegebenen C-Programmteile in einen Datenpfad ohne vorherige Veränderungen übersetzbar. Eine dieser Situationen tritt auf, wenn eine Schleife nur einen lesenden Zugriff auf eine Variable enthält. Innerhalb dieser Schleife ist diese Variable dann als Konstante anzusehen. Ein Problem tritt dann auf, wenn diese Art von Variablen ganz normal durch die DFG-Erzeugung gehandhabt werden (Bild 4.20a). In diesem Fall nämlich wird ein Token für die Variable `y` nur einmal innerhalb der Schleifenausführung erzeugt. Dadurch steht sie nachfolgenden Schleifeniterationen nicht mehr zur Verfügung.

Die in COMRADE verwendete Lösung für dieses Problem ist das Einfügen einer Zuweisung (`y=y`) in die betroffene Schleife. Der DFG wird dann auch mit einem Multiplexer für y erzeugt (Bild 4.20b). Somit wird für jede neu begonnene Schleifeniteration auch ein neues Token für `y` erzeugt.

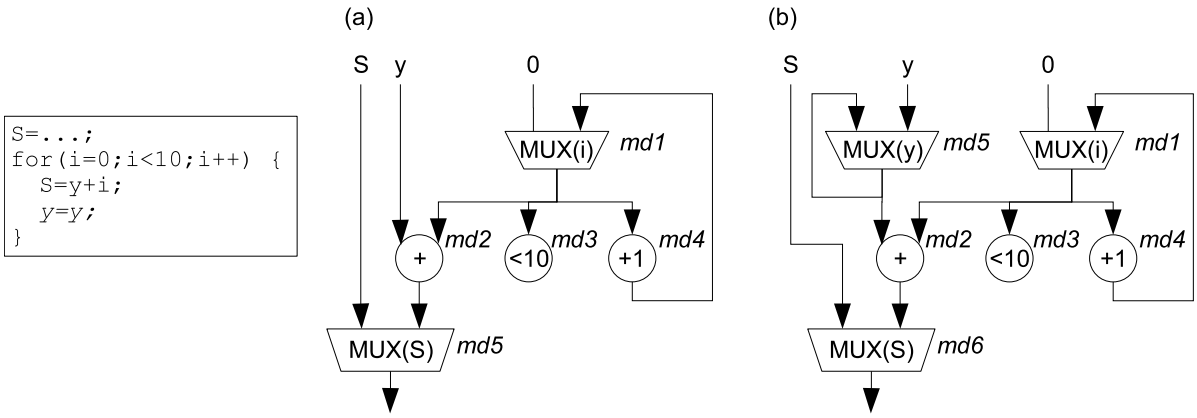


Bild 4.20 Behandlung von Lesezugriffen, (a) ohne `y=y`, (b) mit `y=y`

Die Lösung mit dem zusätzlichen Einbau einer Anweisung hat den Vorteil, dass sie auf einer hohen Darstellungsebene des Programms vollzogen werden kann. Auf der anderen Seite verbraucht diese Lösung mehr Platz auf dem Ziel-RL durch das zusätzliche Register. Auch können alle von `y` abhängigen Berechnungen erst dann ausgeführt werden, wenn der Multiplexer den Wert dafür ermittelt hat.

Eine weitere Lösung des Problems liegt in der Handhabung einer nicht veränderten Variablen innerhalb einer Schleife durch den Kontroller. Im Beispiel könnte der Kontroller solange für `y` solange ein Up-Token erzeugen, wie md_3 auch ein Up-Token erzeugt, also true ist.

4.6 Einbau des Kontrollmechanismus in HW-Module

Während der Implementation der Kontrollererzeugung in COMRADE hat sich gezeigt, dass der Kontroller nicht zusätzlich zum DFG erzeugt werden muss, da er den gleichen strukturellen Aufbau wie der DFG hat. Weiterhin ist das nachträgliche Erzeugen des Kontrollers sehr kompliziert und damit fehleranfällig in der Implementation. Vorteilhaft für eine spätere Implementation wäre die Einbindung von HW-Modulen und zugehörigen Kontrollregistern in ein komplexeres *HW-Makromodul*. Dieses sollte auch wie die HW-Module selbst generisch erzeugbar sein, um es den Gegebenheiten (Anzahl der Vorgänger, Anzahl der Nachfolger, Art des Moduls) anpassen zu können. Diese HW-Module könnten dann aneinandergefügt werden und müssten neben Daten- auch Kontrollinformationen austauschen.

Der Vorteil einer solchen Realisierung wäre in erster Linie eine Vereinfachung der Handhabung durch COMRADE. Auf der anderen Seite können auch an COMRADE anschließende Programme bessere Ergebnisse beispielsweise bei der Platzierung und Verdrahtung erzielen. Alle eng miteinander in Beziehung stehenden Objekte sind in den HW-Makromodulen zusammengefasst und nur noch durch Verdrahtung miteinander verbunden. Ein Platzierungsalgorithmus kann diese Informationen für schnellere Schaltungen verwenden.

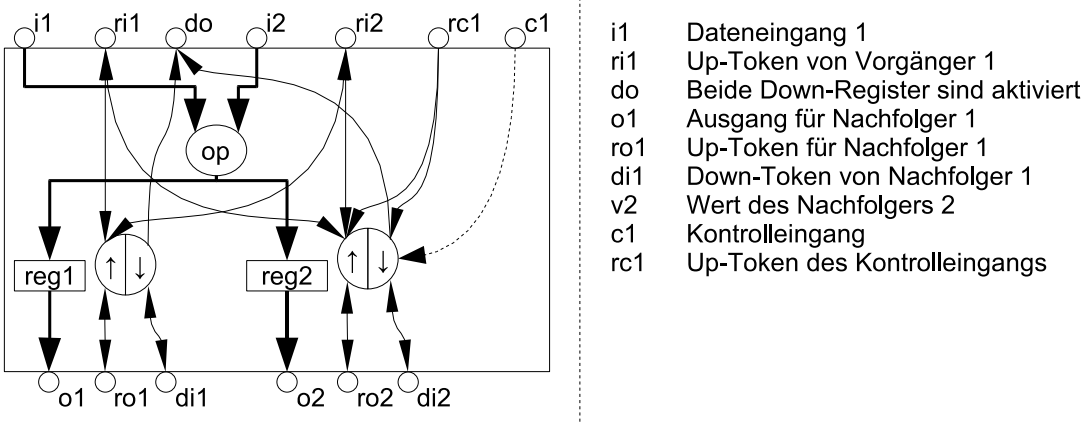


Bild 4.21 Einbindung von Steuerregistern in HW-Makromodule

Ein HW-Makromodul sollte für jeden Nachfolger ein separates Register enthalten. Damit können Schwierigkeiten behoben werden, welche durch die Weitergabe des berechneten Werts unter unterschiedlichen Kontrollflussbedingungen auftreten (Bild 4.21). Die Kontrollinformationen werden innerhalb des HW-Makromoduls gegenüber den Dateninformationen unterschiedlich behandelt (rc1, c1). Die gesamte Logik für die Steuerung der Token-Register ist jeweils in den HW-Makromodulen lokal vorhanden.

4.7 Ergebnisse

Ein wichtiger Faktor für die Anwendbarkeit des neu eingeführten Kontrollers sind die auf der HW verbrauchten Ressourcen. Sind diese zu groß, ist die Anwendung des Kontrollers für reale Projekte nicht zu empfehlen. Tabelle 4.22 zeigt für das Benchmark-Programm versatility

für die in Datenpfade übersetzten Schleifen (Spalte 1) die Anzahlen von HW-Module, von enthaltenen Daten-, Kontroll- und Speicherabhängigkeiten sowie die Anzahl der im Kontroller integrierten Register. Eines dieser Register besteht jeweils aus zwei auf der HW realisierten Flipflops (Up- sowie Down-Register).

Zuerst fällt die hohe Anzahl der Summe der Abhängigkeiten gegenüber der Anzahl der Register auf. Eigentlich sollten diese beiden Anzahlen gleich sein, da für jede Abhängigkeit ein Token zur Verfügung steht. Der Grund für die von diesen Anzahlen abweichenden Messwerte liegt darin, dass der Datenpfad in der Realisierung nicht nur HW-Module enthält, welche auf Logikressourcen wie CLBs abgebildet werden. Vielmehr sind auch Verdrahtungsmodule enthalten. Dadurch erhöhen sich natürlich die Anzahlen der HW-Module, da beispielsweise aus einer Datenabhängigkeit zwischen zwei HW-Modulen bei Einfügen eines Verdrahtungsmoduls zwei Abhängigkeiten (Kanten im DFG) entstehen.

Weiterhin ist erkennbar, dass neben den Speicherabhängigkeiten zwischen zwei HW-Modulen auch Daten- oder Kontrollabhängigkeiten zwischen den gleichen HW-Modulen bestehen. Das ist z.B. beim Kernel hufenc.12 erkennbar, in dem zwischen den sechs Speicherzugriffen nur vier Speicherabhängigkeiten bestehen. Mindestens fünf wären hier aber auf Grund der Regeln zum Aufbau der SSA-Form zu erwarten, wenn alle Speicherzugriffe hintereinander ausgeführt werden müssten.

Die Ergebnisse zeigen, dass bei realen Programmen die Anzahl von Registern nicht die maximal mögliche Anzahl erreicht. Die maximale Anzahl ist direkt von den Anzahlen der Abhängigkeiten im Datenpfad abhängig.

Theoretisch könnten Datenpfade erzeugt werden, in denen jedes HW-Modul von jedem anderen HW-Modul durch eine Daten-, Kontroll- oder Speicherabhängigkeit miteinander verbunden sind (Vollständiger gerichteter Graph). Damit würden sich bei n HW-Modulen $n*n$ Kanten und somit auch Register ergeben, da eine Kante von jeweils einem der n HW-Module auf n HW-Module zeigen kann.

In den Programmen wird diese obere Grenze aber dadurch begrenzt, dass die Operatoren in C maximal drei Operanden besitzen können. Außerdem werden berechnete oder aus dem Speicher gelesene Werte größtenteils nicht parallel für weitere Berechnungen und die Erzeugung von Kontrollabhängigkeiten verwendet. Somit überschreitet die Anzahl der Register die Anzahl der HW-Module im Beispiel auch nicht um den Faktor drei.

Kernel	HW-Module	Speicher-module	Daten-abh.	Kontroll-abh.	Speicher-abh.	Register
versatility						
read_image.11	12	2	36	11	2	29
Block_quantize.32	26	1	76	24	1	59
Block_RLE_encode.46	31	1	80	32	1	79
Block_RLE_encode.34	78	6	229	64	6	184
entropy_encode.14	7	1	24	10	3	23
hufenc.12	24	6	66	19	4	54
fcdf22.28	27	4	72	19	3	57
fcdf22.21	40	9	121	20	8	89
fcdf22.14	28	4	75	20	3	60
adpcm						
main.70	109	4	350	96	4	278
G721						
update.126	12	2	46	15	2	39
update.119	42	11	172	43	12	126
pegwit						
gfInit.52	11	2	45	14	2	37
gfInit.44	18	2	74	19	2	53
gfAdd.51	20	4	76	20	4	58
gfReduce.23	30	8	122	30	6	95
gfMultiply.86	28	5	107	28	4	81
gfMultiply.59	17	4	62	18	3	52
gfSmallDiv.38	29	5	110	30	3	84
gfAddMul.64	19	5	72	26	4	62
gfAddMul.52	29	6	119	29	5	86
gfAddMul.39	16	6	62	19	5	52
capacity						
get_freq.16	10	3	40	13	2	34
printbin16.8	6	0	15	9	0	17
printVHDL.29	4	0	12	7	0	13
printVHDL.22	6	1	23	11	3	24
create_input_file.14	8	0	18	11	0	21
create_output_file.32	6	0	15	9	0	17
create_output_file.25	6	1	23	11	3	24

Tabelle 4.22 Größe des Kontrollers aus Datenpfaden

4.8 Erweiterungsmöglichkeiten

Die Implementierung der Datenpfad-Erzeugung kann die an sie gestellten Anforderungen zurzeit in vollem Umfang erfüllen. Doch sind während der Implementierung von darauf aufbauenden Compilerschritten und während des Tests einige Verbesserungsmöglichkeiten aufgefallen.

Zum einen wurden für die Implementierung Verdrahtungsknoten zwischen HW-Module eingefügt, wenn die Verbindung zwischen diesen nicht durch einen einfachen Bus mit einer festen Breite beschrieben werden kann. Diese Verdrahtungsknoten fügen die aus der Bitbreitenreduktion (Abschnitt 5.3) erhaltenen Informationen in den DFG ein. Zu einem Problem werden diese nicht nur wegen der Verfälschung von Testergebnissen, da sie als Knoten betrachtet werden. Vielmehr werden durch sie auch einige Algorithmen erschwert, da beispielsweise für die Erzeugung des Kontrollers nur die Abhängigkeiten zwischen den HW-Modulen interessant sind. Die Verdrahtungsknoten müssen immer umständlich übersprungen werden. Zur Verbesserung der Probleme sollten die Verdrahtungsknoten entfernt und stattdessen deren Informationen den Kanten beigefügt werden (CE_03).

Die Einbindung von Speicherabhängigkeiten in den Datenpfad ist bisher noch nicht optimal gelöst worden. Bei der derzeitigen Implementation wird noch davon ausgegangen, dass jeder Speicherzugriff jeden anderen beeinflussen kann. Dadurch entstehen aber unnötige Speicherabhängigkeiten im DFG. Werden nun die schon für COMRADE verfügbaren Speicherabhängigkeitsanalysen verwendet, so kann ein Teil der Abhängigkeiten weggelassen werden, da diese in der Realität nicht vorhanden sind. Dadurch wird eine höhere Parallelität bei Speicherzugriffen erreicht, da auf unabhängige Speicherbereiche konkurrierend zugegriffen werden kann (CE_04).

Der DFG von COMRADE unterstützt mehrere Arten von HW-Modulen (Abschnitt 4.2.1). Da diese auf verschiedene Weise auf unterschiedliche Arten angesteuert werden müssen, wird der Aufbau des Kontrollers kompliziert. Zur Vereinfachung wurden schon die HW-Module mit variabler Laufzeit auf nur eine Art der Ansteuerung beschränkt. Diese muss aber bei der Erzeugung der Datenpfad-Beschreibung für die auf COMRADE folgenden Werkzeuge an die eigentlichen HW-Module angepasst werden.

Um jedes HW-Modul auf eine einheitliche Weise durch den Kontroller von COMRADE ansteuern zu können, sollte für jede Art der HW-Module eine umschließende Logik generiert werden, welche der im folgenden beschriebenen Schnittstelle genügt (Bild 4.23a, CE_05). Diese Schnittstelle regelt jeweils die Übergaben von Daten zu und aus einem HW-Modul. So können Daten nur an ein HW-Modul übergeben werden, wenn es dieses an den Kontroller signalisiert (taken ist false). Erhält der Kontroller das Signal, so teilt er dem HW-Modul mit, dass die Daten am Eingang gültig sind (start wird true). Wurden diese von dem HW-Modul eingelesen oder verarbeitet (je nach Art des HW-Moduls), wird dies wiederum dem Kontroller mitgeteilt (taken wird true).

Am Ausgang eines HW-Moduls muss der gleiche Mechanismus vorhanden sein. Hierbei wird die Übergabe eines Datums aus einem HW-Modul dadurch eingeleitet, dass der Kontroller die Bereitschaft dazu signalisiert (finish ist false). Daraufhin reagiert das HW-Modul mit dem Setzen des Signals für die Gültigkeit des Ausgangs (done wird true). Wird der am Ausgang anliegende Wert nicht mehr benötigt, so wird dies durch den Kontroller mitgeteilt (finish wird true).

Dieser auf den ersten Blick komplizierte Mechanismus wird aber je nach Art des HW-Moduls dadurch reduziert, dass verschiedene, vom HW-Modul erzeugte Signale nur durch Verdrahtung oder einfache Logik erzeugt werden. Umschließt man beispielsweise ein HW-Modul mit fester Laufzeit mit der oben geforderten Schnittstelle (Bild 4.23b), dann wird nur ein Flipflop zusätzlich benötigt. Weiterhin können durch die Vereinheitlichung auch Register des Kontrollers eingespart werden.

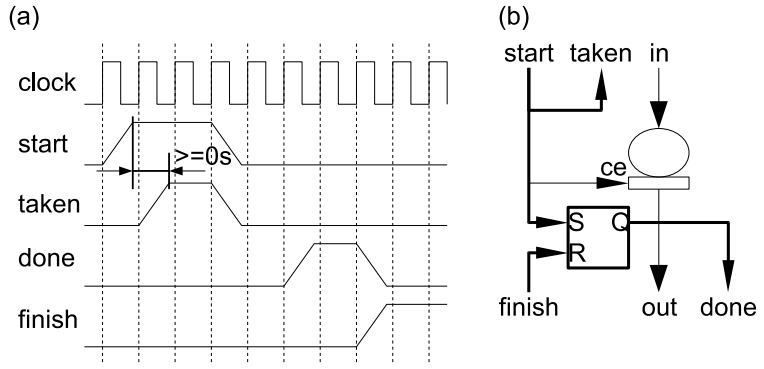


Bild 4.23 Einheitliche Ansteuerung von HW-Modulen, (a) Signalverlauf, (b) Beispiel

4.8.1 Integration von IP-Blöcken

Für viele Anwendungen (Video-Bearbeitung, Ver-/Entschlüsselung) werden vorgefertigte und optimierte HW-Implementierungen (IP-Blöcke) angeboten. Diese sind meist besser als die durch COMRADE aus C-Quelltext automatisch erstellten HW-Implementierungen. Deshalb wäre es vorteilhaft, verfügbare IP-Blöcke in die Programmerstellung für Adaptive Computersysteme integrieren zu können.

Da die IP-Blöcke oft keine standardisierten Schnittstellen haben und auch ihre Funktion oft nicht durch eine einfache Beschreibungssprache wiedergegeben werden kann, wäre eine automatische Integration in COMRADE nur mit sehr hohem Aufwand machbar. Der Compiler müsste erkennen, welche Teile des Quellcodes mit der Funktion eines IP-Blocks übereinstimmen. Auf Grund dieser Schwierigkeit sollten IP-Blöcke von Hand im Quelltext durch den Programmierer instanziiert und angesteuert werden.

Eine mögliche Art der Integration besteht darin, einen IP-Block als Funktionsaufruf zu integrieren (CE_06). Anhand des Namens der Funktion würde dann COMRADE erkennen,

dass es sich um einen IP-Block handelt. Für diesen müssten in einer Datenbank oder Beschreibungsdatei alle für die Verwendung nötigen Informationen wie das Schnittstellenprotokoll sowie HW-relevante Daten enthalten sein. COMRADE könnte dann den IP-Block in den erzeugten Datenpfad integrieren. Weiterhin muss der Programmierer neben der Instanziierung des IP-Blocks auch die Ansteuerung desselben durch Einfügen von Funktionen in den Quellcode beschreiben. Diese teilen dem Compiler dann mit, wann welche Eingangsdaten des IP-Blocks zu schreiben und wann welche Ausgangsdaten zu lesen sind.

4.8.2 Verbesserung des Kontrollers

In der derzeitigen Implementierung von COMRADE wurde gezeigt, dass eine automatische Erzeugung eines Kontrollers für die Datenpfad-Steuerung mit besonderen Eigenschaften möglich ist. Der von CORMADE erzeugte Kontroller lässt nach Möglichkeit die HW-Module in Pipelines ausführen und ist dazu fähig, spekulativ begonnene Berechnungen zu unterbrechen. Leider ist der generierte Kontroller noch nicht perfekt. Eine Erweiterung der HW-Implementierung der Datenpfade ist die Integration von Kontroller-Komponenten in HW-Module und die damit verbundene Erzeugung von Makro-HW-Modulen (Abschnitt 4.6, CE_07).

Aber es sind auch Verbesserungen in den Aktivierungs- und Deaktivierungsbedingungen der Token-Register denkbar. So kann beispielsweise zurzeit ein Token-Register nur dann aktiviert werden, wenn es nicht schon aktiviert ist. Diese Eigenschaft war für den Test des Kontrollers nützlich, da damit die Funktion besser nachvollzogen werden konnte. Nachteilig aber ist, dass beim Verlassen eines Up-Registers auch der Wert im Ausgangs-Register eines HW-Moduls nicht mehr benötigt wird. Das Ausgangs-Register könnte eigentlich schon wieder mit einem neu berechneten Wert geladen werden. Da der Kontroller aber darauf wartet, dass das Up-Register nicht betreten ist, verzögert sich der Ablauf an dieser Stelle um einen Takt. Würde man stattdessen darauf reagieren, ob das Up-Register gehalten werden soll ({FSM}_STATE_UP_HOLD in Bild B.1), dann könnte die HW-Implementation noch schneller laufen (CE_08).

Kapitel 5

High-Level-Optimierungen für Datenpfade

Im Rahmen des COMRADE-Projekts wurde eine Auswahl von Optimierungen implementiert, um deren Auswirkungen auf die entstehende HW zu testen. Die Optimierungen arbeiten dabei nur auf den Regionen im CFG, welche als HW-Kandidaten ausgewählt wurden. Durch die nur teilweise Bearbeitung der Anwendungsprogramme wird Laufzeit des Compilers eingespart. Außerdem erleichtert das Behalten der ursprünglichen Form des Programms im SW-Teil die Fehlersuche, da die Programmstrukturen weitestgehend erhalten bleiben.

Die folgenden Abschnitte behandeln die für die Optimierungen grundlegenden Analysen und stellen die in COMRADE implementierten Optimierungen sowie deren Einfluss auf die resultierende HW vor.

5.1 Analysen von Speicherabhängigkeiten

Neben den Kontrollflussbedingungen und einfachen Datenflussabhängigkeiten innerhalb des CFG in SSA-Form ist vor allem eine genaue Analyse von Datenabhängigkeiten durch Speicherzugriffe (Feldzugriffe, dereferenzierte Pointer) grundlegend für die Erzeugung einer effizienten HW-Realisierung. Der Grund liegt darin, dass viele C-Programme nicht für eine HW-Realisierung geschrieben wurden. Somit fehlen wichtige, für die HW-Erzeugung relevante Daten wie die mögliche parallele Ausführbarkeit von Speicherzugriffen. Solche Zusatzinformationen sind oft nur in speziell für die HW-Synthese angepassten C-Sprachderivaten mit Anweisungen für parallele Verarbeitung [Celo03] zu finden. Analysen und Optimierungen sollen automatisch Verbesserungspotential in solchen Programmen finden.

Für COMRADE stehen zwei Analyseverfahren für Speicherabhängigkeiten zur Verfügung. Die für das SUIF2-System verfügbare Steensgard-Analyse [Stee96] findet neben anderen Pointer-Alias-Analysen [YoHR99, FoFA00, HiPo01] Abhängigkeiten zwischen verschiedenen Zugriffen auf den Speicher oder Variablen mittels Pointern oder Referenzen heraus.

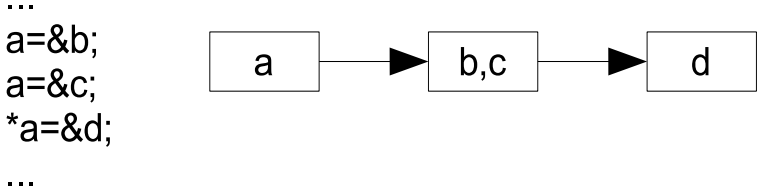


Bild 5.1 Abhängigkeitsgraph von abstrakten Speicherstellen [FoFA00]

Bei dieser Analyse wird ein Abhängigkeitsgraph zwischen einzelnen Variablenmengen aufgebaut (Bild 5.1). Die Variablenmengen sind die Knoten des Abhängigkeitsgraphen. Diese repräsentieren denselben Speicherbereich im Hauptspeicher. Eine Kante zwischen zwei Knoten entsteht dann, wenn die Variablen im Startknoten die Speicherbereiche der Variablen im Zielknoten referenzieren. Wird der durch eine im Startknoten referenzierte Speicherbereich verändert, so können sich die Werte der im Zielknoten befindlichen Variablen ändern. In Bild 5.1 werden beispielsweise der Variablen `a` in unterschiedlichen Anweisungen die Referenzen für die Variablen `b` und `c` zugewiesen. Es entsteht eine Kante von {a} nach {b,c}. Würde man auf die durch `a` referenzierte Speicherstelle zugreifen, so würde eine dieser Variablen verändert. Durch `*a=&d` wird der durch `a` referenzierten Speicherstelle (`b` oder `c`) die Referenz auf `d` zugewiesen ({b,c}→{d}). Erkennbar ist hier, dass das Auslöschen der Bindung von `a` an die Referenz an `b` nicht durch die Anweisung `a=&c` ihre Gültigkeit verliert. Die Steensgard-Analyse wertet also keine Kontrollflussinformationen aus.

Eine weitere für COMRADE verwendete Analyse ist die Abhängigkeitsanalyse zwischen Feldzugriffen. Bei dieser wird versucht, bei Feldzugriffen innerhalb von Schleifen Zugriffsmuster in Abhängigkeit von Iterationsvariablen herauszufinden. Dabei ist es für das Analyseprogramm möglich, Richtungsvektoren wie auch Abstandvektoren zwischen einzelnen Feldzugriffen herauszufinden [Much97].

Eine Voraussetzung für das Bestimmen von Abhängigkeiten ist das Vorhandensein und Erkennen einer Iterationsvariablen innerhalb eines C-Programms. Die Iterationsvariablen müssen für COMRADE einen linearen Iterationsraum besitzen, ihren Wert von einer Schleifeniteration zu einer folgenden also nur um einen konstanten Wert erhöhen. Mit Hilfe dieser werden dann die Zugriffsfunktionen analysiert. Diese können wiederum durch unterschiedliche Arten von Funktionen beschrieben werden. Zur Einfachheit müssen auch diese nur linear von der Iterationsvariablen abhängig sein. Alle anderen werden als nicht identifizierbar angenommen. Als Auswirkung dessen können in diesen Fällen Abhängigkeiten zwischen zwei Speicherzugriffen erkannt werden, auch wenn sie nicht existieren. Da aber Zugriffe mit linearen Zugriffsfunktionen sehr häufig in Programmen vorkommen, sind die Effekte nicht gravierend für COMRADE.

Da COMRADE auf normalen C-Programmen arbeitet, können Analyseansätze wie zur vollständigen Zerlegung von Zugriffsräumen benutzte [LiLa97, KiRD00] nicht oder nur in begrenztem Rahmen praktiziert werden. Hierzu müssen die Zugriffsräume immer vollständig durch den Compiler analysierbar sein. Dies ist aber nicht immer machbar, da C-Programme auch nichtlineare Iterationsräume enthalten können, Schleifen nicht immer perfekt geschachtelt sind und lineare Zugriffe durch Feldzugriffe nicht immer auftreten. Für viele Optimierungen ist es darüber hinaus sogar nicht notwendig, diese Räume vollständig zu kennen. Hierbei sind beispielsweise Abhängigkeitsvektoren zur Beschreibung vollkommen ausreichend.

Für die Analyse in COMRADE wird die OMEGA-Bibliothek [Pugh92] verwendet, welche mittels Presburger Arithmetik Abhängigkeitsanalysen durchführen kann [Moll04]. In einem C-Programm werden hierzu die Zugriffsfunktionen aller Feldzugriffe aufgestellt. In

diesen Funktionen sind die Iterationsvariablen inklusive ihrer Schrittweiten von einer Schleifeniteration zur nächsten sowie unteren und oberen Grenzen enthalten, soweit diese Informationen ermittelbar sind. Die Zugriffsfunktionen von jeweils zwei Feldzugriffen werden in Zusammenhang gebracht und auf Informationen bezüglich Abhängigkeiten zwischen verschiedenen Iterationen untersucht. Des Weiteren werden die Zugriffsfunktionen in COMRADE auch mit Gleichungen für Verzweigungen innerhalb von C-Programmen erweitert, da diese die Abhängigkeiten zwischen Feldzugriffen oft erheblich verändern können.

Für die Abhängigkeitsanalyse werden in COMRADE C-Programme auf der Darstellungsebene eines CFG bearbeitet. Der Algorithmus nutzt dazu die speziellen Eigenschaften eines Dominatorbaums aus (Beispiel 5.2c). Die Bearbeitung auf Grund des Dominatorbaums wurde gewählt, da:

1. alle in einer Schleife enthaltenen Kontrollflussblöcke durch den Schleifenkopf und
2. alle durch eine Verzweigung beeinflussten Kontrollflussblöcke durch den Kontrollflussblock mit der Verzweigungsanweisung

dominiert werden. Deshalb können die in den Schleifenköpfen und Verzweigungen gesammelten Daten bei Preorder-Bearbeitung des Dominatorbaums immer an die beeinflussten Blöcke weitergegeben werden. Dabei muss man aber bei Verzweigung darauf achten, welche Relation man in die abhängigen Knoten weitergibt. Diese sind von der Bedingung in der Verzweigung abhängig und deswegen in der Regel für jeden der dominierten Knoten unterschiedlich.

Im Beispiel 5.2 müssen die im Kontrollflussblock kb_6 enthaltenen drei Speicherzugriffe analysiert werden. Diese befinden sich innerhalb von zwei ineinander geschachtelten Schleifen und einer Verzweigung. Wenn nun der Algorithmus den Dominatorbaum bearbeitet, so kann er aus den Schleifenköpfen in kb_1 und kb_3 ermitteln, dass

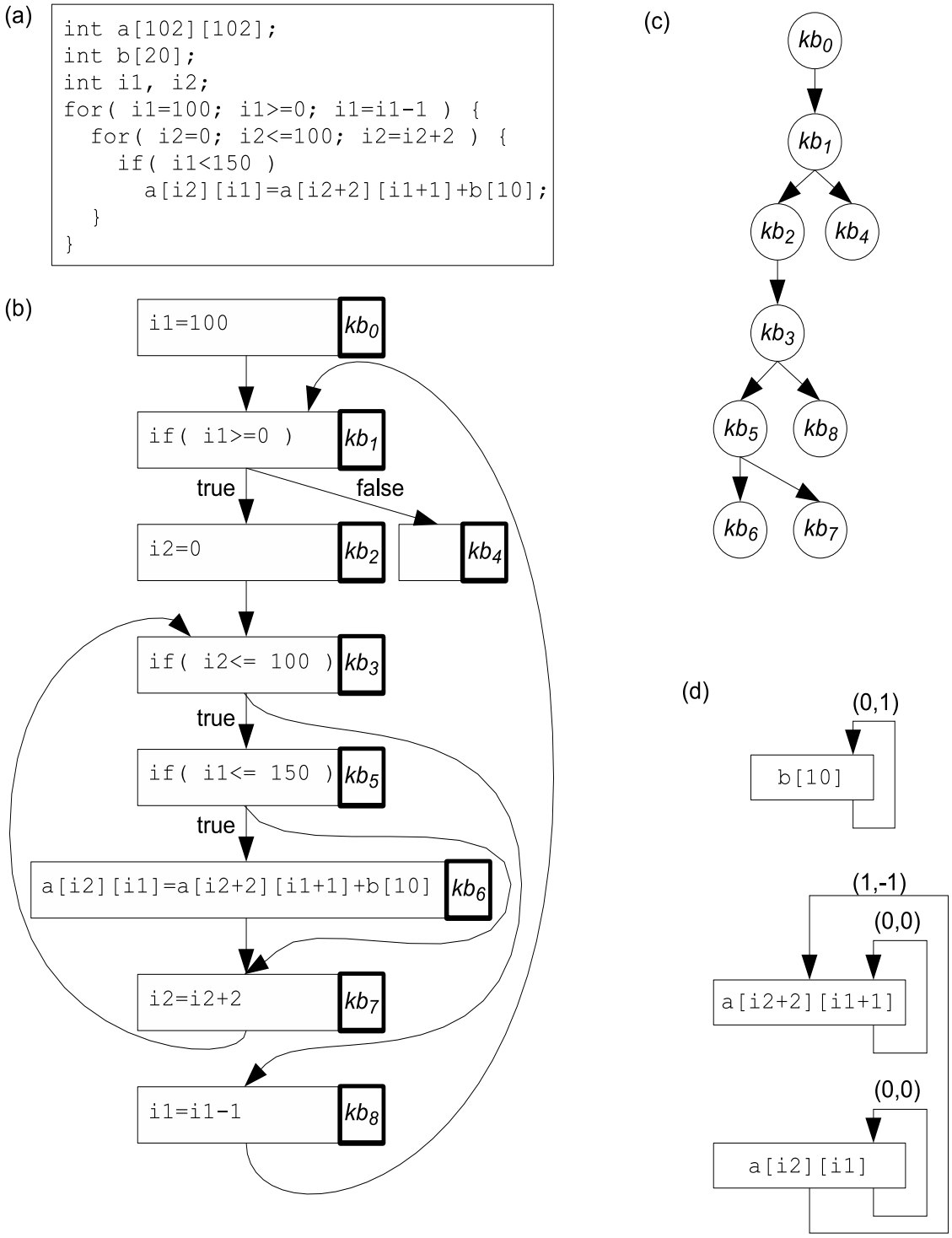
1. die Iterationsvariable i1 im Wertebereich von 100 bis 0 mit Schrittweite –1 und
2. die Iterationsvariable i2 im Wertebereich von 0 bis 100 mit Schrittweite 2 iterieren.

Weiterhin kann die Bedingung in kb_5 in eine Relation umgewandelt werden. Werden die so ermittelten Relationen und Gleichungen mit den Zugriffsfunktionen der Feldzugriffe kombiniert, so stellt man fest, dass die Bedingung in kb_5 die Zugriffsfunktionen nicht einschränkt. Sie hat also keine Bedeutung.

Die aus den Schleifenköpfen und Bedingungen ermittelten Relationen werden nun mit den Zugriffsfunktionen der Feldzugriffe und den Feldrelationen (untere, obere Grenzen) verknüpft. Um nun die Beziehungen zwischen allen Feldzugriffen herauszufinden, werden die Zugriffsfunktionen mit gemeinsamen Basisvariablen (`a`, `b`) miteinander kombiniert und der OMEGA-Bibliothek zur Analyse übergeben. Aus dem Analyseergebnis sind die in Beispiel 5.2d aufgezeigten Abhängigkeiten ableitbar. Zur Einfachheit halber werden in der Zeichnung nur die Distanzvektoren angegeben.

Zwischen den einzelnen Feldzugriffen bestehen als erstes Selbstabhängigkeiten. Außerdem existiert für die beiden Zugriffe mit der Basisvariablen `a` auch noch eine so genannte Lese-Schreib-Abhängigkeit mit einem Distanzvektor (1,-1). Die erste Komponente ist in diesem Beispiel eine Aussage über Abhängigkeiten von der Iterationsvariablen `i1` die zweite

eine über `i2`. Für den Zugriff auf den Speicher kann man so ableiten, dass abhängig von `i1` auf die gleiche Speicherstelle zuerst durch `a[i2][i1]` und eine Schleifeniteration in der äußeren Schleife später auch durch `a[i2+2][i1+1]` zugegriffen wird. Dementsprechend wird in der inneren Schleife abhängig von `i2` auf die gleich Speicherstelle zuerst von `a[i2+2][i1+1]` und dann von `a[i2][i1]` zugegriffen.



Beispiel 5.2 (a) for-Schleifen mit Feldzugriffen, (b) CFG, (c) Dominatorbaum und (d) Abhängigkeiten

Die Distanz- und Richtungsvektoren könnten für Optimierungen des DFG (Weglassen von Speicherabhängigkeiten in Abschnitt 4.2.5) oder eigene Optimierungsschritte wie die skalare Ersetzung oder Schleifenumformungen in COMRADE verwendet werden (Abschnitt 5.4).

5.2 Constant-Propagation

Die Aufgabe von Constant-Propagation liegt in der Ersetzung von Variablen mit konstantem Wert durch die enthaltene Konstanten. Weiterhin können auf diese Weise Teile eines C-Programms herausgefunden werden, welche auf Grund von Verzweigungen mit konstanten Bedingungen nicht ausgeführt werden. Wegen der zu erwartenden Reduzierung der Größe der HW-Kandidaten wurden **Simple-Constant-Propagation** [Appe98] und **Sparse-Conditional-Constant** [WeZa91, Appe98] für COMRADE implementiert [Wint03]. Diese beiden Algorithmen werden in COMRADE für eine Ausführung hintereinander vorgesehen und beinhalten weiterhin eine als **Copy-Propagation** bezeichnete Optimierung. Durch diese werden Zuweisungen von einer Variablen zu einer anderen eliminiert. Die Constant-Propagation-Algorithmen sind weiterhin durch ein mögliches Zusammenfassen von Operationen auf Konstanten zu einer Konstanten erweitert (`c=a+b` mit a=11 und b=31 ergibt `c=42`).

Wichtig für die Ausnutzung der Konstanten zur Reduzierung der Größe von HW-Kandidaten ist der Einsatz von geeigneten, anschließenden Optimierungen sowie eines Modulgenerators für die Abbildung von Operatoren des C-Programms auf die HW-Module. Optimierungen wie die Bitbreitenreduktion (Abschnitt 5.3) integrieren Konstanten in Operatoren und können dadurch die Bitbreite von Verdrahtungen sowie ganze Operatoren einsparen. Außerdem können die Konstanten während der Anfrage an den Modulgenerator von diesem in die generierten Module eingebaut werden. Die Verwendung von vorgefertigten HW-Bausteinen für die Abbildung von SW-Operatoren auf HW-Module oder das Module-Mapping [CaHW00] ist dazu nur mit einem hohen Aufwand in der Lage.

5.2.1 Funktionsweise von Constant-Propagation

Constant-Propagation arbeitet auf der SSA-Form (Abschnitt 4.2.2), welche die Laufzeit und außerdem die Komplexität des Algorithmus verringert. Grundlegend sind die von der SSA-Form erzeugten Datenflussabhängigkeiten. Diese werden aktiv zum Feststellen von Auswirkungen des Definitionszustands einer Variablen auf andere Variablen benutzt.

Zum Anfang werden alle Variablen als undefiniert angenommen. Ein iterativer Algorithmus sammelt dann aus den Zuweisungen an die Variablen Erkenntnisse über den Definitionszustand von Variablen. Ändert sich der Zustand einer dieser Variablen, so werden die Auswirkungen auf andere Variablendefinition untersucht. Der iterative Algorithmus endet, wenn sich keine Änderungen des Definitionszustands für die Variablen mehr ergeben. Eine Analyse von in Verzweigungen verwendeten Variablen kann nun erkennen, ob die Bedingungen in Verzweigungen konstant sind. So nicht mehr erreichbare Kontrollflussblöcke werden

dann gelöscht. Außerdem können auch die Verzweigungen aus dem CFG entfernt werden, da sie nur noch einen Nachfolger-Kontrollflussblock auswählen.

Bei Sparse-Conditional-Constant werden weiterhin die in den weggelassenen Blöcken befindlichen Zuweisungen als nicht mehr existent markiert und die Auswirkungen auf abhängige Zuweisungen verarbeitet. Dadurch können unter bestimmten Voraussetzungen bessere Ergebnisse als durch Simple-Constant-Propagation erzielt werden.

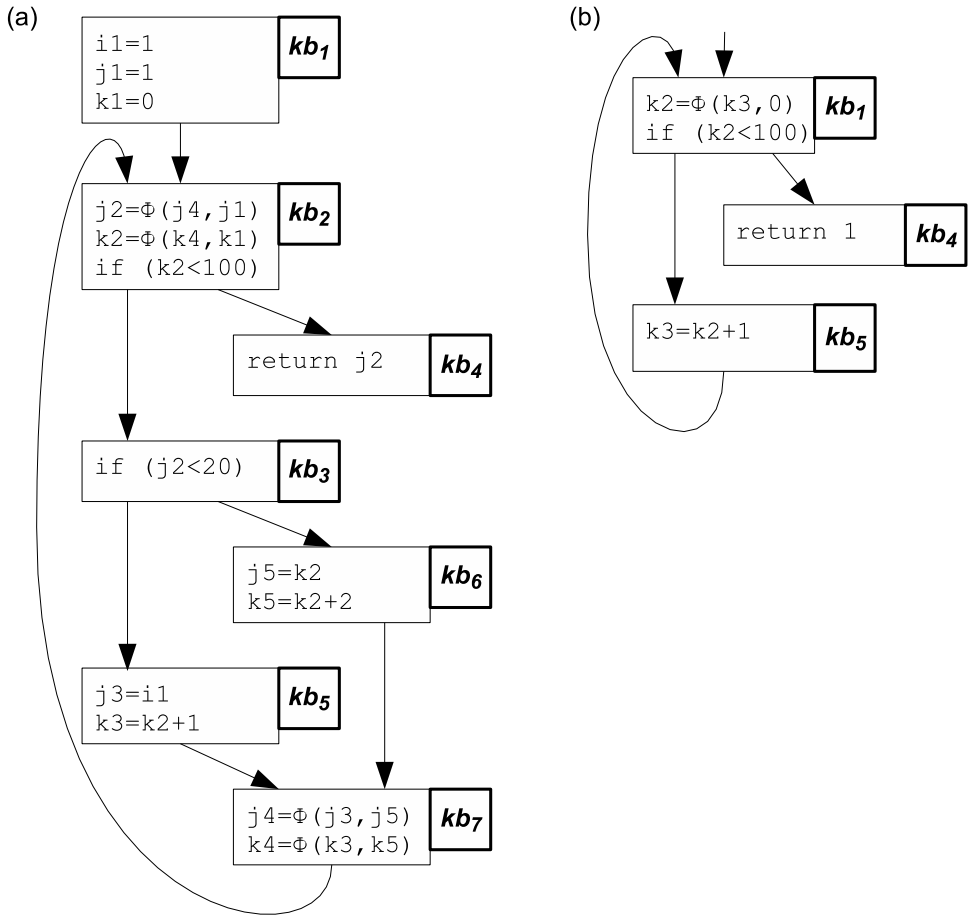


Bild 5.3 Beispiel für Sparse-Conditional-Constant

Das Beispielprogramm (Bild 5.3a) enthält Variablen, denen Konstanten zugewiesen wurden (`i1`, `j1`, `k1`). Der Algorithmus Sparse-Conditional-Constant kann nun erkennen, dass durch Zuweisungen der konstanten Variablen an andere die Bedingung in Kontrollflussblock kb_3 niemals falsch werden kann und somit kb_6 nie ausgeführt wird. Der Kontrollflussblock mit der Bedingung ist nun ohne Bedeutung und kann gelöscht werden. Außerdem entfallen die Zuweisungen an die Variablen `j5` und `k5`, wodurch wiederum `j4` nur den Wert `j3` und `k4` nur den Wert von `k3` annehmen können. Durch Einsetzen der konstanten Werte der so als konstant erkannten Variablen können alle Variablen bis auf `k2` und `k3` durch Konstanten ersetzt werden. Nach Weglassen aller nicht mehr benötigten Anweisungen und Kontrollflussblöcke ergibt sich ein deutlich kleineres Programm (Bild 5.3b).

5.2.2 Auswirkungen auf die Größe von Datenpfaden

Werden die Konstanten der Bitbreitenreduktion übergeben, so kann diese vor allem bitweise Operationen (AND, OR, XOR, SHIFT) optimieren. Shift-Operationen mit konstanten Kontrollwerten sind sehr gute Kandidaten für Laufzeit- und Flächenoptimierungen auf der Ziel-HW. Der Grund hierfür liegt in der Reduzierung eines shift-HW-Moduls (Bild 5.4a) durch Einbeziehung von Konstanten in Verdrahtung (Bild 5.4a). Der shift-Operator kann also vollkommen eingespart werden.

Des Weiteren sparen auch andere Operationen durch die Einbindung von Konstanten Laufzeit und Fläche. Eine als Strength-Reduction [BaGS93] bezeichnete Optimierung kann beispielsweise eine Multiplikation (Bild 5.4c) durch Einbeziehung einer Konstanten in Operationen mit geringerem Ressourcenverbrauch (Bild 5.4d) konvertieren. Diese Optimierung ist als eigener Compilerschritt oder im Modulgenerator integrierbar.

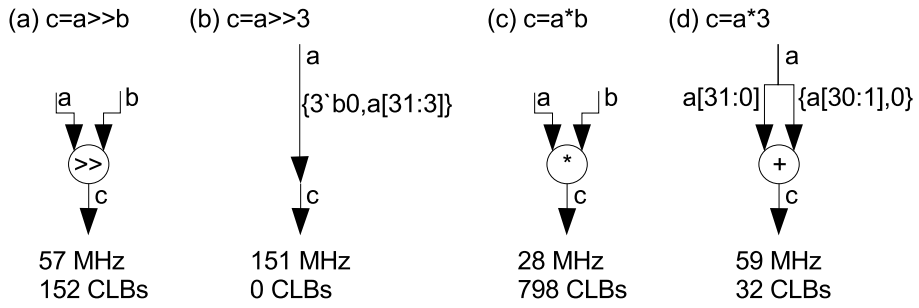


Bild 5.4 HW-Einsparung durch Einbinden von Konstanten

Vorteilhafter ist die Integration in den Modulgenerator, da dieser auf Grund von besserem Wissen über die Struktur des Ziel-RL nicht nur die Operatortypen sondern auch den Aufbau der HW-Module selbst optimieren kann. Führt der Compiler die Optimierung durch, so können oft aus einem HW-Modul mehrere andere entstehen. Information über die Beziehung zwischen diesen geht während der Compilierung verloren und muss bei Bedarf beim Platzieren und Verdrahten wieder neu erschaffen werden. Generiert aber der Modulgenerator ein angepasstes Modul, so bleiben die Informationen erhalten.

5.3 Bitbreitenreduktion

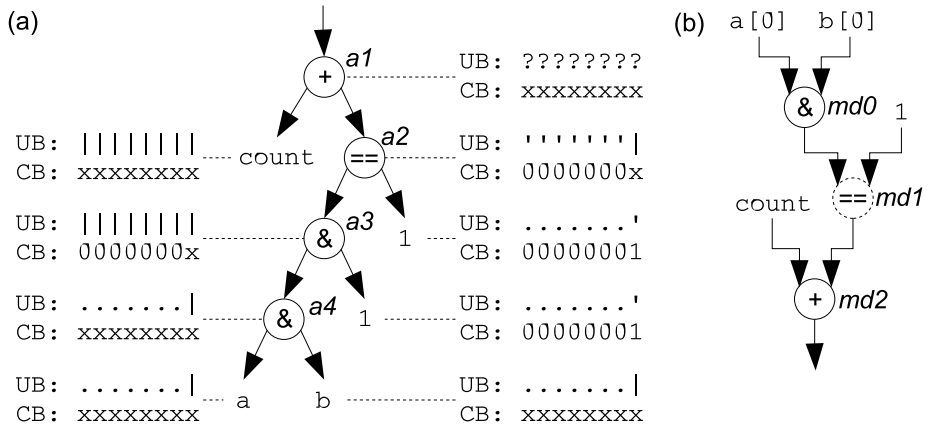
Bei der Programmierung von C-Programmen für Prozessoren können die Programmierer oft nur auf die im ANSI-C festgelegten Standardtypen für Variablen zurückgreifen. Oft wird so aber auch der größtmögliche Datentyp verwendet, da bei der Programmierung eines Algorithmus der wirklich benutzte Wertebereich einer Variablen oft nicht bekannt ist. Außerdem ist eine bitgenaue Festlegung der Bitbreite von Datentypen durch die von Prozessoren verwendeten Standardbitbreiten nicht möglich.

Bei der HW-Ausführung eines Algorithmus ist die bitgenaue Bearbeitung natürlich möglich, wenn man bestimmen kann, auf welche Bits man die Operatoren beschränken kann. Oft können dabei die Bitbreiten erheblich reduziert werden. Die damit verbundene Einsparung von Verdrahtungs- und HW-Modul-Breiten sowie die mögliche Laufzeitreduktion von HW-Modulen beeinflussen entscheidend die Qualität der HW-Implementationen.

Eine Überarbeitung der C-Programme per Hand ist dabei in vielen Fällen möglich, erfordert von den Programmierern aber Wissen über die bearbeiteten Algorithmen und Zeit. Deshalb wurde in COMRADE ein Compilerschritt zur automatischen Reduktion von Bitbreiten in den HW-Kandidaten implementiert. Gegenüber anderen Ansätzen mit der Verwaltung von verwendeten Wertebereichen von Variablen [StBA00] werden in COMRADE für jedes Bit Informationen über benutzte und konstante Bits geführt. Der Vorteil dieser Vorgehensweise liegt darin, auch einzelne Bits innerhalb von Bussen zu entfernen oder auf einen konstanten Wert zu setzen. Außerdem werden so auch Optimierungen wie das Aufbrechen von Carry-Chains in Addierer-HW-Modulen möglich.

5.3.1 Verwenden von Bitmasken

Bei der Ausführung der Bitbreitenredution in COMRADE [Mück03] werden während der Analyse von Anweisungen zwei verschiedene Bitmasken für jeden in einem AST auftretenden Operator gespeichert. Diese werden dann als Grundlage für die Optimierung von Operationen und Verdrahtung verwendet.



UB: Bitmaske für verwendete Bits
(. = nicht verwendet, | = Bit des Operators wird verwendet, ' = Bit der Konstante wird verwendet)
CB: Bitmaske für konstante Bits
(x = nicht konstant, 0 = konstant 0, 1 = konstant 1)

Beispiel 5.5 Bitbreitenreduktion im Ausdruck `count+(((a&b)&1)==1)`
(a) AST mit Bitmasken, (b) optimierter DFG

Die erste Bitmaske speichert für jeden Operator konstante Bits sowie deren Werte. Im Beispiel (a) ist erkennbar, dass für die Konstanten im AST diese Bitmaske die Repräsentation als Bitmuster enthalten. An den Operatoren werden die Bitmasken der Operanden miteinander verknüpft und somit Optimierungspotential ermittelt. Beim AND-Operator *a3* beispielsweise kann sich am Ausgang nur noch das unterste Bit ändern. Alle anderen Bits sind durch die Einbeziehung der Konstanten auf Null festgelegt.

Die Bitmaske für die verwendeten Bits gibt an, welche Bits des Ergebnisses eines Operators verwendet werden. Bei der Verwendung wird außerdem zwischen der Verwendung des Ergebnisses der Operation oder der Verwendung einer Konstanten unterschieden.

5.3.2 Funktionsweise der Analyse

Die einfache Variante der Bitbreitenreduktion in COMRADE nutzt nur die lokal vorhandenen Informationen an separaten ASTs. Jeder AST wird bei der Analyse jeweils zweimal bearbeitet. Dabei werden die Konstanten auf zwei unterschiedliche Arten zum Erkennen von nicht benötigten Bits verwendet.

Im ersten Durchlauf in Postorder-Reihenfolge der Operatoren im AST werden die einzelnen Operatoren mit Bitmasken für konstante Bits versehen. Alle nicht als konstant erkannten Bits müssen im optimierten AST immer noch durch die im Operator beschriebene Operation erzeugt werden. Für alle anderen Bits werden nur die erkannten Konstanten als Ergebnis zurückgeliefert. Diese konstanten Bits brauchen aber nicht mehr durch die Operanden dieser Operatoren berechnet werden. Nach dem ersten Durchlauf sind Informationen über verwendete Bits aber noch nicht an den Operanden vorhanden.

Der zweite Durchlauf in Preorder bestimmt nun für jeden Operator, welche Bits noch berechnet werden müssen. Diese Informationen werden dann an die Operanden weitergegeben. Wenn keiner der Operanden mehr benötigt wird, kann der Durchlauf des AST hier abgebrochen werden.

Die Ergebnisse der beiden Baumdurchläufe für Beispiel 5.5 zeigen, dass von den Konstanten nur die untersten Bits verwendet werden, da nur diese für die Darstellung der Konstanten notwendig sind. Durch die Analyse der benutzten Bits im AND-Operator *a3* kann nun festgestellt werden, dass nur das Ergebnis der Verknüpfung im untersten Bit wichtig ist. Dieses Ergebnis spiegelt sich in der Bitmaske für die verwendeten Bits wieder. Somit kann der zweite AND-Operator *a4* auch auf einen 1-Bit-Operator verkleinert werden. Außerdem müssen von den Erzeugern der Variablen `a` und `b` nur noch 1-Bit-Leitungen an den Operator geführt werden.

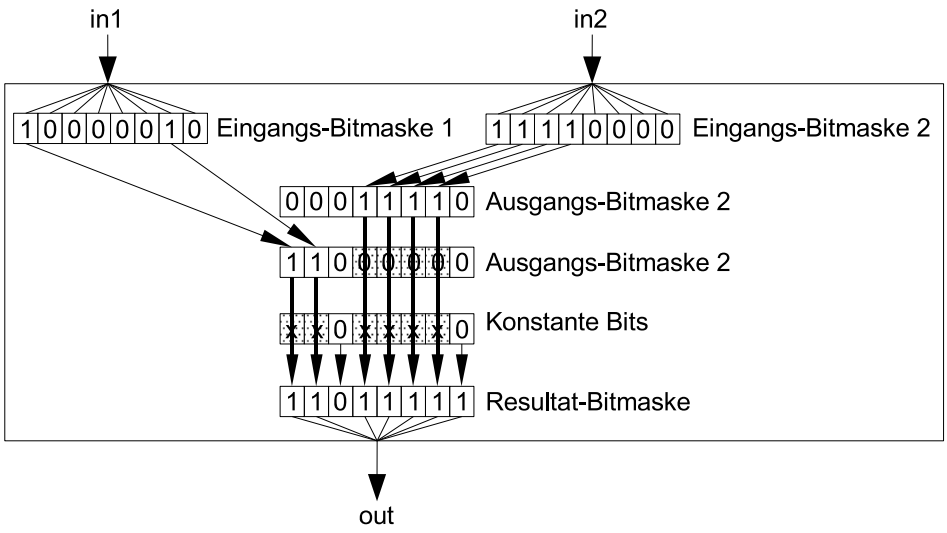
Auf Grund der Analyse kann im Beispiel durch Weglassen und Optimieren von Operatoren auf einen DFG mit weniger Ressourcenverbrauch reduziert werden (Beispiel 5.5b). Hier konnte ein AND-HW-Modul eingespart sowie bei zwei weiteren HW-Modulen die Bitbreite reduziert werden. Nur die Addition muss noch auf der vollen Bitbreite arbeiten, kann aber durch ein vereinfachtes Modul ersetzt werden, da höchstens eine Eins addiert werden muss.

Weiterhin kann auch der Vergleicher weggelassen werden, da sein Ergebnis mit dem des AND-HW-Moduls identisch ist.

5.3.3 Verdrahtungsoperatoren

Die Bitbreitenreduktion findet auf den ASTs statt. Damit die Ergebnisse auch Auswirkungen auf die erzeugten DFGs haben, müssen die Ergebnisse der Analyse auch innerhalb des ASTs eingefügt werden. Sie stehen dann nachfolgenden Compilerschritten zur Verfügung. Um alle Möglichkeiten für die Einbeziehung der Ergebnisse der Bitbreitenreduktion bereitzustellen, wurden spezielle Verdrahtungsoperatoren (Beispiel 5.6) eingeführt, um die Auswirkungen der Bitbreitenreduktion auf den AST wie auch Typkonvertierungen (z.B. `signed int` nach `unsigned int`) in den AST aufnehmen zu können.

Bei diesen Operatoren, welche als Versionen mit einem bzw. zwei Eingängen in COMRADE implementiert worden sind, wird zuerst jedem Eingang eine Bitmaske zugeordnet. Diese Bitmaske wählt die vom jeweiligen Eingang verwendeten Bits aus. Außerdem wird für jeden Eingang eine Ausgangs-Bitmaske benötigt, welche für die Zuordnung der vom Eingang gewählten Bits den Ausgangsbits des Operators zuordnet. Hierbei kann es vorkommen, dass bestimmte Bits des Ausgangs durch keine der Ausgangs-Bitmasken an Bits der Eingänge zugeordnet werden. Diese Bits werden durch im Operator gespeicherte konstante Bits aufgefüllt. Für das Weglassen einzelner Bits innerhalb des Ausgangs wurde in der Implementierung noch eine weitere Bitmaske eingeführt (Resultat-Bitmaske).



Beispiel 5.6 Verdrahtungsoperator mit zwei Eingängen

5.4 Skalare Ersetzung

Wie in fast jeder Rechnerarchitektur stellt die Kommunikationsbandbreite der rechnenden Einheiten zum Speicher einen limitierenden Faktor bei der Laufzeit dar. Zu diesem Zweck

wurden verschiedene Optimierungen entwickelt, welche sich auch für die Realisierung in HW eignen [AlKe02].

So können mehrere Schleifen zusammengefasst werden, um gleiche Feldzugriffe in unterschiedlichen Schleifenkörpern zu vereinen (*Loop-Fusion*). Außerdem kann auch die Umordnung der Hierarchie von Schleifen zur Reduzierung von Speicherzugriffen führen (*Loop-Interchange*). Erfolge werden weiterhin mit dem Abrollen einer äußeren Schleife und dem Zusammenfassen der dann entstehenden Kopien von inneren Schleifen (*Unroll-and-Jam*) erreicht.

In COMRADE wurde eine weitere Optimierung implementiert: Die **Skalare Ersetzung** (*Scalar-Replacement*) [CaCK90] Der Grundgedanke hierbei ist das Halten von durch Feldzugriffe gelesenen Werten in temporären Variablen über mehrere Schleifeniterationen. Da durch diese Optimierung viele temporäre Variablen und Zuweisungen erzeugt werden können, ist sie für Prozessoren mit einer begrenzten Anzahl von Registern nicht immer ideal. Die Werte der temporären Variablen müssen bei einer zu großen Anzahl immer wieder aus den Registern des Prozessors in einen verfügbaren Cache oder schlimmstenfalls den Arbeitsspeicher geschrieben und auch von dort wieder geladen werden. Die Einsparung von Feldzugriffen wird hierdurch wieder zunichte gemacht. Abhilfe schafft bei der Optimierung für Prozessoren eine Begrenzung der Erzeugung von temporären Variablen durch eine Heuristik.

In einer Realisierung auf einem RL hingegen stehen größere Mengen an Registern zur Verfügung. Durch die Nutzung dieser kann der zu erzielende Gewinn durch die Skalare Ersetzung voll ausgenutzt werden.

Grundlage für die Optimierung ist eine Analyse der Feldzugriffe durch den Compiler. Die erkannten Abhängigkeiten zwischen Feldzugriffen werden als Grundlage für die Erkennung von Zugriffen verwendet, welche:

1. zuletzt auf eine Position im Speicher zugreifen oder
2. bei Fehlen eines Schreibzugriffs auf eine Position im Speicher zuletzt von dieser lesen.

Die anderen Zugriffe werden dann als Generatoren bezeichnet, wenn von ihnen eine Schreib-Lese-Abhängigkeit oder Lese-Lese-Abhängigkeit ausgeht. Mit der Zwischenspeicherung der durch die Generatoren erfolgten Zugriffe in temporäre Variablen und deren Verwendung in anderen Feldzugriffen kann jeweils mindestens ein Feldzugriff eingespart werden.

Erkennt die Skalare Ersetzung in einem Programm (Beispiel 5.7a) bearbeitbare Abhängigkeiten zwischen Feldzugriffen, so werden für diese Feldzugriffe temporäre Variablen eingeführt (Beispiel 5.7b). Die Anzahl der temporären Variablen hängt dabei von der Anzahl der durch sie zu überbrückenden Schleifeniterationen ab. Für die Feldzugriffe mittels der Basisadresse von `a` beispielsweise werden drei temporäre Variablen eingeführt (`a_i0`, `a_i1`, `a_i2`). Dies ist notwendig, weil zwischen den Zugriffen `a[i+2]` und `a[i]` eine Distanz von zwei Schleifeniterationen liegt. Die in diesen Feldzugriffen gelesenen Werte müssen zwischengespeichert werden. Durch diese Variablen werden die Lesezugriffe auf die Speicherstelle `a[i]` innerhalb der Schleife durch Kopiervorgänge am Ende des Schleifenkörpers (`a_i0 = a_i1`, `a_i1 = a_i2`) ersetzt. Somit werden die auf eine Speicherstelle geschrie-

benen Werte so lange von Iteration zu Iteration getragen, bis sie von einer Leseoperation verwendet werden.

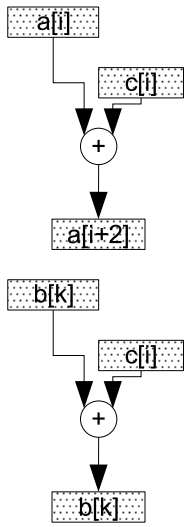
(a)

```
for( i=1; i<n; i++ ) {
  a[i+2] = a[i] + c[i];
  b[k] = b[k] + c[i];
}
```

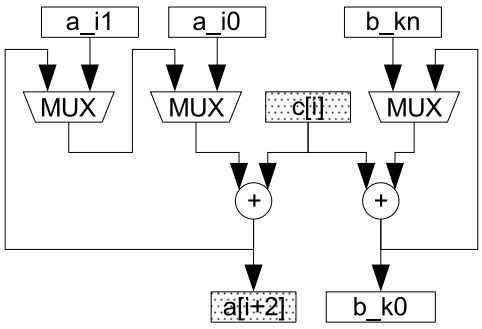
(b)

```
c_i0 = c[1];
a_i2 = a[1] + c_i0;
a[3]  = a_i2;
b_k0 = b[k] + c_i0;
a_i1 = a_i2;
b_kn = b_k0;
c_i0 = c[2];
a_i2 = a[2] + c_i0;
a[4]  = a_i_2;
b_k0 = b_kn + c_i0;
a_i0 = a_i1;
a_i1 = a_i2;
b_kn = b_k0;
for( i=3; i<n; i++ ) {
  c_i0  = c[i];
  a_i2  = a_i0 + c_i0;
  a[i+2] = a_i_2;
  b_k0  = b_kn + c_i0;
  a_i0  = a_i1;
  a_i1  = a_i2;
  b_kn  = b_k0;
}
b[k] = b_k0;
```

(c)



(d)



Beispiel 5.7 Skalare Ersetzung: Auswirkungen auf die HW-Realisierung

Die HW-Implementierung der optimierten Version (Beispiel 5.7d) besitzt gegenüber der unoptimierten Version (Beispiel 5.7c) weniger Zugriffe auf den Speicher (zwei statt sechs). Dies spart Rechenzeit, da die Speicherzugriffe meist der limitierende Zeitfaktor für die Laufzeit einer HW-Implementierung sind. Bedenkt man weiterhin, dass die Speicherzugriffe im aktuellen System noch nicht für eine parallele Ausführung vorgesehen sind, ist die Laufzeitersparnis somit erheblich. Man kann erkennen, dass die Einsparung auf zwei Arten erreicht wurde:

1. Die lesenden und schreibenden Zugriffe mittels `b[k]` konnten auf Grund der Unabhängigkeit von `k` von der Iterationsvariablen `i` vor bzw. hinter die Schleife verschoben werden. Die Berechnung des auf `b[k]` nach der Schleife zu schreibenden Werts erfolgt mittels der temporären Variablen `b_k0` und `b_kn`.
2. Die für das Feld `a` eingeführten temporären Variablen bilden in der HW-Implementierung ein in den Multiplexern verstecktes Schieberegister.

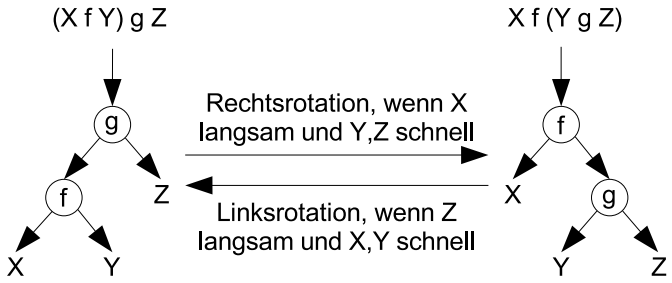
Die durch die Optimierung erhöhte Anzahl von zwischen HW und SW auszutauschenden Variablen hat keinen negativen Einfluss auf die Laufzeit der HW-Implementierung. Die reduzierte Laufzeit durch die Einsparung von Feldzugriffen ist im Allgemeinen viel größer als die für die Kommunikation zwischen HW und SW verbrauchte Zeit.

5.5 Baumhöhenreduktion

Eine weitere in COMRADE implementierte Optimierung ist die **Baumhöhenreduktion** von Operationsbäumen [Krah03]. Bei dieser Optimierung wird versucht, die Berechnungszeit von ASTs in Hinblick auf die HW-Implementierung zu verringern. Dies ist notwendig, da die als Eingabe für COMRADE verwendeten Programme nicht für eine HW-Realisierung vorgesehen sind und deshalb auch nicht so modelliert sind, dass die Operationsbäume optimal in HW realisiert werden könnten. So würde ein Programmierer den Ausdruck a+b+c+d nicht in einer HW-freundlichen Version als (a+b)+(c+d) schreiben, da dies für die Ausführung auf dem Prozessor keinen Laufzeitgewinn bringen würde.

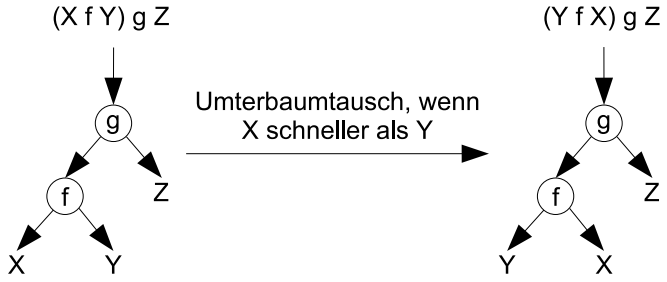
In der Realisierung in HW dagegen können mehrere Operationen auch gleichzeitig ausgeführt werden. Somit könnte a+b und c+d parallel berechnet und somit gegenüber der ursprünglichen Version die Zeit für die Berechnung einer Addition gespart werden.

Die Baumhöhenreduktion führt auf dem AST lokale Umformungen aus, welche Assoziativität oder Kommutativität der im AST enthaltenen Operatoren voraussetzt. Sind zwei Operatoren f und g im AST assoziativ ((X f Y) g Z = X f (Y g Z)), so können diese zur Reduzierung der Laufzeit im aus dem AST entstehenden Datenpfad rotiert werden (Beispiel 5.8).



Beispiel 5.8 Baumrotationen

Zur besseren Ausbalancierung der ASTs können vor einer Rotation die Unterbäume vertauscht werden (Beispiel 5.9), wenn X schneller als Y ist. Diese Umformung ist nur bei Kommutativität von f möglich.



Beispiel 5.9 Unterbaumtausch

Die beschriebenen Operationen werden beim rekursiven Bearbeiten des ASTs an den Ausdrücken vollzogen. Zur Unterstützung dient neben dem AST ein Kostenbaum mit gleicher Struktur. Bei Experimenten mit COMRADE hat sich herausgestellt, dass die Reihenfolge der Bearbeitung der Operatoren im AST eine Rolle für das erzielte Ergebnis spielt. Mit einer Bearbeitung eines AST mit acht Summanden in Postorder-Reihenfolge konnte beispielsweise nicht zu einem dreistufigen AST optimiert werden. Andererseits sind aber auch Ausdrücke nicht immer bei einer Preorder-Bearbeitung optimierbar. Darüber hinaus arbeitet dieser auch nicht in linearer Zeitabhängigkeit von der Anzahl Operatoren im AST. Eine Kombination von Postorder- und Preorder-Reihenfolge in der Bearbeitung stellt einen guten Kompromiss zwischen der Anwendung dieser Heuristik und der Berechnung der optimalen Lösung dar.

5.6 Praktische Ergebnisse

Um die Auswirkung der oben beschriebenen Algorithmen beurteilen zu können, wurden diese auf die schon aus den anderen Kapiteln bekannten Benchmark-Programme angewandt.

Programm	Eingesparte Bits gesamt/pro Operator	Durchschnittliche Ausgangsbreite davor/danach	Entfernte Operatoren	Durchschn. eingesparte/ Gesamt-Ressourcen (CLB)
versatility	1840/8	28/28	36	12/1409
capacity	255/5	26/25	76	0/182,9
adpcm	560/12	22/19	80	174/1568
pegwit	6208/15	26/25	229	13,18/941,33
g721	1344/14	29/27	24	32/838,72
des	3248/14	29/28	109	80/1505,33

Tabelle 5.10 Eingesparte Bits durch Bitbreitenreduktion

Wird die Bitbreitenreduktion ohne vorhergehendes Constant-Propagation durchgeführt, dann sind bei den Beispielprogrammen schon gute Ergebnisse erzielbar (Tabelle 5.10). In der zweiten Spalte sind hierzu die insgesamt im Programm eingesparten Bits sowie normiert auf die Anzahl der Operatoren im Programm angegeben. Dabei ist die Anzahl der Operatoren für die Normierung die vor der Optimierung. In der dritten Spalte sind die durchschnittlichen Bitbreiten der Ausgänge der Operatoren jeweils vor und nach der Optimierung zu sehen. Die vierte Spalte gibt die Anzahl der eingesparten Operatoren an. Dies sind größtenteils Konvertierungsoperatoren, die in der HW-Realisierung durch Verdrahtung ersetzt werden können. In der letzten Spalte sind die durchschnittlich eingesparten Ressourcen gegenüber dem durchschnittlichen Ressourcenverbrauch der Datenpfade angezeigt. Bis auf capacity verwenden auch die HW-Implementierungen weniger Ressourcen. Die Daten bei capacity sind darauf zurückzuführen, dass bei Einsparungen der Bitbreite auch die Reduzierungen der Ausgangsbitbreite von booleschen Operatoren eingehen. In C hat eine boolesche Variable eine Breite

von 32 Bit. Das HW-Modul hat aber in der optimierten wie auch in der unoptimierten Version nur eine Ausgangsbitbreite von einem Bit. Die nicht genutzten 31 Bit werden der Bitbreitenreduktion gutgeschrieben.

Bei versatility ist zu erkennen, dass die eingesparten Bits nur durch herhausgefallene Operatoren begründet sind. Die Ausgangsbitbreite der Operatoren hat sich nicht verkleinert. Weiterhin ist erkennbar, dass sich die durchschnittliche Ausgangsbitbreite bei der in COMRADE implementierten einfachen Bitbreitenoptimierung schon um bis zu zwei Bit reduzieren kann. Bessere Ergebnisse sind natürlich durch die Einbeziehung von Informationen möglich, welche über die Ebene von ASTs hinausgehen.

Tabelle 5.11 zeigt die Ergebnisse von Constant-Propagation auf die Benchmark-Programme und die Bitbreitenreduktion. Neben dem Namen des Benchmark-Programms in der ersten Spalte wird in der zweiten Spalte die Anzahl der erkannten Konstanten angegeben. Diese Anzahl wurde durch die Ausführung von Sparse-Conditional-Constant und Simple-Constant-Propagation hintereinander erkannt. Die dritte Spalte führt die während der Ausführung der Optimierungen erkannten Kopieranweisungen (`a=b;`) auf. In der vierten Spalte ist die Anzahl der durch neue Werte ersetzten Variablen gegeben. Die letzte Spalte zeigt die Auswirkungen von Constant-Propagation auf die Ergebnisse der Bitbreitenreduktion.

Programm	Erkannte Konstanten	Kopier-anweisungen	Ersetzte Variablen	Ergebnis der Bitbreitenreduktion, Ausgangsbreite ohne/mit (Bits) : Einsparung ohne/mit (CLB)
versatility	24	29	73	28/27 : 12/13,75
capacity	0	9	10	25/24 : 0/0
adpcm	8	17	37	19/18 : 174/222
pegwit	2	15	27	25/25 : 13,18/122,72
g721	0	4	5	27/26 : 32/40
des	8	28	33	28/28 : 80/580,67

Tabelle 5.11 Ersetzte, als konstant oder kopiert erkannte Variablen

An den Werten kann man erkennen, dass vor allem der versatility-Benchmark stark von diesen Optimierungen profitiert. In den anderen Programmen sind entweder keine oder nur geringe Optimierungsaussichten erkennbar.

Die durch die Optimierung entfernten Kopieranweisungen haben keine Auswirkung auf die resultierende HW-Implementierung, da diese nur in Verdrahtung übersetzt werden. Vielmehr reduzieren sie die Laufzeit des Compilers, da sie nicht mehr bearbeitet werden müssen.

Bei der Auswirkung von Constant-Propagation auf die Bitbreitenreduktion kann man erkennen, dass diese die erkannten Variablen in ihre Arbeit mit einbeziehen kann. Gegenüber den Ergebnissen der Bitbreitenreduktion ohne vorheriges Constant-Propagation kann so in jedem Benchmark-Programm noch einmal bis zu einem Bit in der durchschnittlichen Aus-

gangsbitbreite der HW-Module eingespart werden. Bei versatility kann man aus der letzten Spalte von Tabelle 5.11 erkennen, dass die durchschnittliche Ausgangsbitbreite von 28 auf 27 Bits durch die Hinzunahme von Constant-Propagation reduzieren konnte. Statt der durchschnittlichen Einsparung von 12 CLBs pro HW-Kandidat sind nun 13,75 CLBs Einsparung möglich geworden. Bei der Einsparung von HW-Ressourcen können aber fast alle Programme von Constant-Propagation profitieren. Beachtenswert ist die Einsparung bei des mit ca. einem Drittel der durchschnittlichen Gesamtressourcen.

Die Baumhöhenreduktion konnte leider in den getesteten Programmen keine starken Optimierungsmöglichkeiten finden (Tabelle 5.12). So konnten in nur zwei der Benchmarks die ASTs so umgeformt werden, dass sie eine Verbesserung der Laufzeit erbrachten. Das lässt den Schluss zu, dass die ASTs in den Programmen oft schon den Anforderungen für eine HW-Implementierung entsprechen. Ein weiterer möglicher Grund schien die Verwendungen von temporären Variablen zu sein. Deshalb wurde ein Compiler-Schritt implementiert, welcher Hilfsvariablen (z.B. `t` in `t=a+b;d=c-t`) entfernte. Aber auch diese Optimierung blieb ohne Erfolg. Die Baumhöhenreduktion ist also für die Optimierung von C-Programmen für die HW-Implementierung, zumindest in unseren Beispielen, nicht relevant. Die durch sie verbrauchte Laufzeit im Compiler ist durch das schlechte Ergebnis nicht zu rechtfertigen.

Programm	Ohne Entfernen temporärer Variablen		Mit Entfernen temporärer Variablen	
	Optimierte ASTs opt./gesamt	Einsparung pro Operator	Optimierte ASTs opt./gesamt	Einsparung pro Operator
versatility	2/296	59*10-10s	2/296	59*10-10s
capacity	0/55	0s	0/42	0
adpcm	0/126	0	0/126	0
pegwit	5/220	59*10-10s	5/169	59*10-10s
g721	0/56	0	0/42	0

Tabelle 5.12 Erfolg der Baumhöhenreduktion

Für die Skalare Ersetzung konnte in den oben angegebenen Programmen nur eine Verbesserung des Quellcodes für versatility festgestellt werden. In diesem Benchmark wurden die 22 im Original enthaltenen Lesezugriffe auf 17 in der optimierten Version reduziert. Außerdem konnten einige der Schreibzugriffe auch aus dem Schleifeninneren vor oder hinter die Schleife verschoben werden. Dadurch wurde die Anzahl der lesenden Zugriffe von 4971151 auf 3008238 um 39,5% und die Anzahl der schreibenden Zugriffe von 2686790 auf 2662265 um 0,9% reduziert. Das compress-Programm wurde hierzu bei beiden Versionen mit der Option lena.pgm aufgerufen.

5.7 Erweiterungsmöglichkeiten

Beim genaueren Betrachten der erzeugten Datenpfade können noch Schwachstellen in der lokalen Optimierung (Peephole-Optimierungen wie Strength-Reduction) von HW-Modulen auftreten. So werden beispielsweise Kombinationen von mehreren unterschiedlichen Operationen, in denen Konstanten auftreten, nicht vollständig optimiert. Somit wird beispielsweise die Abfrage, ob eine Variable gerade oder ungerade ist mit dem Ausdruck `x%2==1` zurzeit durch ein Modulo- und ein Vergleicher-Modul realisiert. Dabei würde hier aber das Abfragen des niederwertigsten Bits der Variablen `x` ausreichen (CE_09).

Neben diesen einfachen Optimierungen sind in COMRADE auch andere Optimierungen noch nicht implementiert worden, welche sich in anderen Projekten für die HW-Synthese als vorteilhaft herausgestellt haben. Das sind beispielsweise Schleifentransformationen wie Loop-Unrolling oder Loop-Tiling. Eine Möglichkeit zur Nutzung dieser fehlenden Optimierungen besteht in der Nutzung des C-Front-Ends und der fehlenden Optimierungen eines anderen Compilers und der Transformation der Zwischendarstellung dieses Compilers in das SUIF2-Format (CE_10). Ein Kandidat für diesen Zweck wäre beispielsweise der Open64-Compiler [Open04].

Eine weitere Schwäche der in COMRADE implementierten Optimierungen ist die Einschränkung ihres Sichtbereichs auf eine Prozedur. Durch die Einbeziehung von globalen Daten oder Daten aus aufrufenden Prozeduren wäre aber das Optimierungspotential noch größer. Vor allem Constant-Propagation würde davon profitieren. Zwar wird dieser Umstand teilweise durch das Inlining von Prozeduren umgangen. Da aber nicht alle Prozeduren in die `main`-Prozedur eingebunden werden, können auch hier die Optimierungen nicht alle Informationen nutzen.

Spekulative Berechnungen werden zurzeit nur aufgrund von Datenabhängigkeiten durchgeführt. Es existieren aber auch Optimierungen, welche in Pipelines Lesezugriffe auf den Speicher in einer Schleifeniteration spekulativ ausführen, wenn noch nicht alle Schreiboperationen der vorhergehenden Schleifen abgeschlossen sind [MaHo00]. Das lohnt sich dann, wenn der Schreibzugriff nicht in jeder Schleifeniteration ausgeführt wird (Beispiel 5.13, Zeile 4), der Lesezugriff einer folgenden Schleifeniteration (Beispiel 5.13, Zeile 3) in einem Datenpfad mit einer Pipeline also meist den richtigen Wert liest. In Erweiterung der in MaHo00 vorgestellten Technik, welche die Leseoperation noch einmal durchführt, wäre aber die Übergabe des geschriebenen Werts an die Operation sinnvoll, welche normalerweise das Ergebnis der Leseoperation bearbeitet.

```
1    for (i = i_init; i < n; i++) {
2        v = A[i];
3        if (v > B[k]) {
4            B[k] = v;
5        }
6    }
```

Beispiel 5.13 Bedingter Feldzugriff

Des Weiteren sind auch im CASH-Compiler [BuGo03] verwendete Optimierungen wie die Verwendung mehrerer unabhängiger Iterationsvariablen und das Einsparen von äquivalenten Speicherzugriffen in COMRADE verwendbar. Gegenüber der dortigen Implementierung wäre aber die Anwendung auf einer höheren Ebene als der des Datenflussgraphen vorteilhaft. Dadurch lassen sich auch noch Änderungen im Kontrollflussgraph wie beispielsweise das Verschieben von Anweisungen vor Schleifen vollziehen, wenn solche Optimierungen ausführbar wurden. Im Datenflussgraph sind die Kontrollflussabhängigkeiten nicht mehr vollständig vorhanden.

Eine weitere nützliche Erweiterung von COMRADE wäre die Einführung von Anweisungen, welche die parallele Ausführung von Programmteilen beschreiben (ähnlich `fork` und `join` in der Hardware-Beschreibungssprache VERILOG). Durch die schon integrierten Analysen wäre es durch neu zu schaffende Compilerschritte möglich, nicht durch Datenabhängigkeiten verbundene Programmteile zu identifizieren und diese dann auf der HW parallel ausführen zu lassen. Die in der Zwischendarstellung eingeführten Anweisungen werden nicht in die Datenpfade integriert. Ihre Aufgabe besteht vielmehr darin, Regionen, deren Datenpfade parallel ausführbar sind, zu einer Region zusammenzufassen und durch den Compilerschritt der DFG-Erzeugung in einen Datenpfad integrieren zu lassen. Dazu könnte die schon implementierte SSA-Form die in [SSHW94] beschriebenen Erweiterungen erfahren.

Kapitel 6

Rekonfigurations-Scheduling

Wie schon in Abschnitt 3.4 angedeutet, sind die in COMRADE erzeugten Datenpfade so klein, dass mehrere von Ihnen gemeinsam zu einer Konfiguration für den Ziel-RL zusammengefasst werden können. Weiterhin ist die Rekonfigurationszeit des RLs noch heute ein Problem für die Nutzung eines ACS zur Beschleunigung von Programmen. Diese im Vergleich zur Laufzeit eines Datenpfads hohen Zeiten verschlechtern das Laufzeitverhalten der durch COMRADE erzeugten Programme.

Zur Verringerung der so entstehenden Verzögerungen wurde in COMRADE ein Compilerschritt implementiert, welcher mehrere Datenpfade zu einer Konfiguration zusammenfasst und somit die während der Programmausführung benötigten Programmstopps auf Grund von Rekonfigurationen minimieren soll.

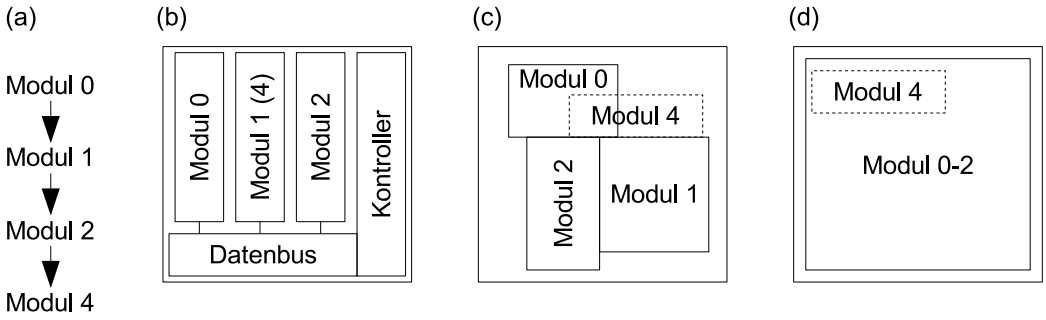
6.1 Verwandte Projekte

Das in COMRADE implementierte Rekonfigurations-Scheduling unterscheidet sich in den an es gestellten Anforderungen und der Vorgehensweise von solchen, welche ein Scheduling während der Laufzeit eines Programms vorsehen. Dabei können diese Verfahren in zwei Gruppen aufgeteilt werden. In der ersten Gruppe sind Form und Größe der für den Ziel-RL übersetzten Datenpfade stark beschränkt [HTKZ04, UHGB04]. Sie können nur an bestimmten Plätzen auf dem Ziel-RL platziert werden (Beispiel 6.1b). Werden die einzelnen Module auf dem RL in der in Beispiel 6.1a gezeigten Reihenfolge benötigt, so muss eines der Module ausgetauscht werden, wenn es nicht mehr benötigt wird. Darüber hinaus wird bei anderen Ansätzen sogar versucht, auch in der Form unterschiedliche Datenpfad-Implementierungen während der Laufzeit auf dem Ziel-RL auf Anforderung zu platzieren [AhBT04] (Beispiel 6.1c).

Bei dem Scheduling in COMRADE spielt die endgültige Platzierung und Verdrahtung der Datenpfad-Implementierungen keine Rolle. Nur die durch die Datenpfade verwendeten Ressourcen werden zu Grunde gelegt. Wichtig dabei ist, die auf dem RL bereitgestellten Ressourcen so gut wie möglich auszunutzen, ohne zuviel zu verwenden (Beispiel 6.1d). Das zu minimierende Gewicht ist dabei die Anzahl der während eines Programmlaufs nötigen Rekonfigurationen.

An dieses Scheduling werden gegenüber der Compile-Zeit-Optimierung von in der Form festgelegten Datenpfad-Geometrien [TeFS99] nicht so hohe Anforderungen gestellt. Gegenüber COMRADE stehen in diesem Projekt als Ausgangsdaten für das Scheduling nicht nur die

durch die Datenpfade verbrauchten Ressourcen sondern auch die Formen der Datenpfade auf dem RL zur Verfügung.



Beispiel 6.1 Laufzeit-Rekonfigurationen mit (a) Reihenfolge zu ladender Datenpfade, (b) gleichförmigen, (c) ungleichförmigen Datenpfaden und (d) nur Modulgrößen

6.2 Ausgangsdaten

Für das Zusammenfassen von Datenpfaden zu größeren Konfigurationen wurde versucht, Heuristiken basierend nur auf einer geringen Anzahl von Informationen zu entwickeln, damit auch die Anwendbarkeit bei größeren Programmen mit deutlich mehr Datenpfaden erhalten bleibt. Ansonsten könnte die Laufzeit der Heuristiken zu stark anwachsen.

Eine wichtige Größe für das Zusammenfassen der Datenpfade ist natürlich der Ressourcenverbrauch. Wird durch eine Konfiguration die Anzahl der auf dem RL verfügbaren Ressourcen überschritten, so kann diese nicht mehr implementiert werden. Die Pfadselektion (Abschnitt 3.2.2) sichert dem Rekonfigurations-Scheduling aber zu, dass der Ressourcenverbrauch jedes Datenpfads die Gesamtressourcen des RLs nicht überschreitet. Eine Aufteilung der Datenpfade in Konfigurationen, bei der jeder Datenpfad in jeweils einer Konfiguration liegt, wäre also auch möglich. Dieser Fall wird als schlechteste Aufteilung der Datenpfade in Konfigurationen angenommen.

Als weitere Daten werden Kontrollflussinformationen (Verzweigungen, Schleifen) sowie Profiling-Daten (Wie oft wird jeder Datenpfad benötigt?) verwendet. Die in Nimble wichtigen Listen über die Reihenfolge der Ladevorgänge von Datenpfaden (*Loop-Traces*, Wann wird jeder Datenpfad verwendet?) werden in COMRADE nur zur Evaluierung der Ergebnisse verwendet. Als Grundlage wird also die Information verwendet, dass beispielsweise Datenpfad eins 20-mal, Datenpfad zwei 25-mal und Datenpfad drei 15-mal ausgeführt wird. Die Reihenfolge (Datenpfad eins, Datenpfad eins, Datenpfad eins, Datenpfad zwei, Datenpfad eins, ...) wird nicht verwendet.

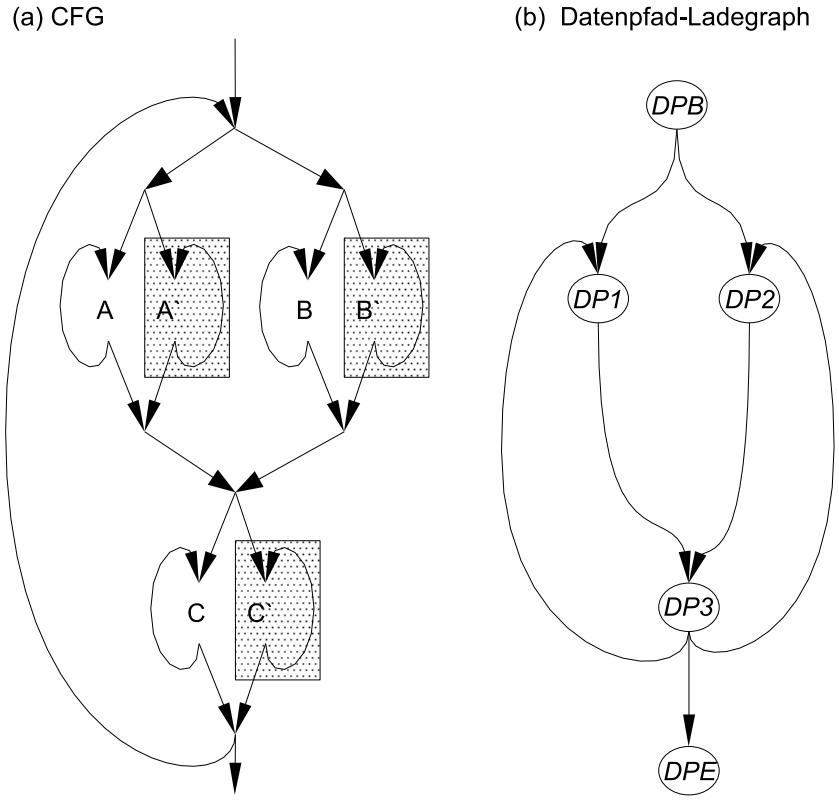
Der Grund hierfür liegt in der Verfügbarkeit dieser Informationen. Sie sind nur durch die Ausführung des zu bearbeitenden Programms ermittelbar. Sie werden in COMRADE zurzeit

parallel zum dynamischen Profiling erlangt. Wird dieses aber nicht benötigt (z.B. wenn ein statisches Profiling vorhanden ist), dann müssten die Loop-Traces trotzdem noch durch die Ausführung ermittelt werden. Wie die Ergebnisse in Abschnitt 6.5 zeigen, ist der Mehraufwand nicht lohnenswert.

6.3 Der Datenpfad-Ladegraph

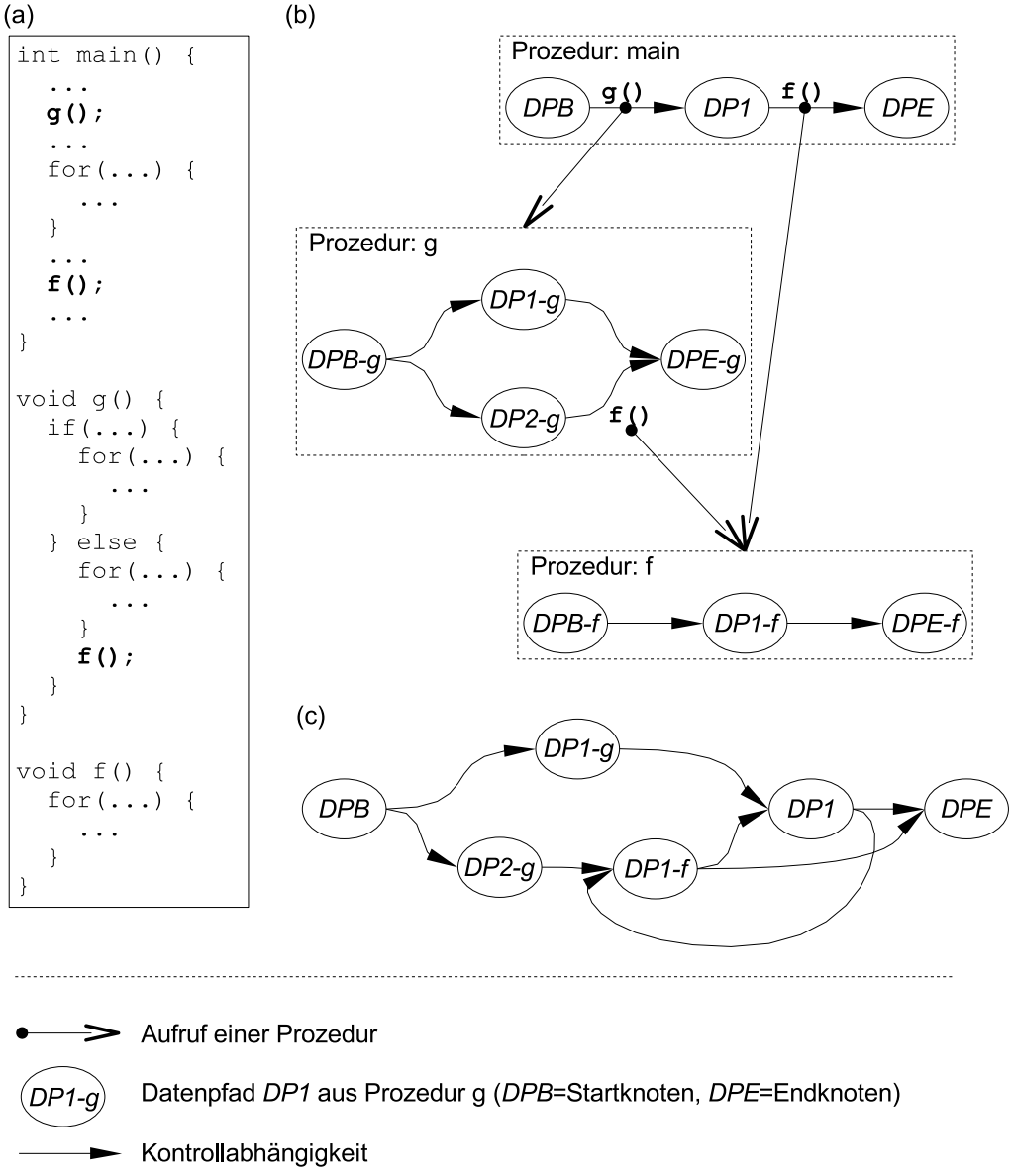
Als Lösung des Problems wird ein **Datenpfad-Ladegraph** erzeugt. Grundlegend für diesen sind die HW-Kandidaten, welche durch die HW-SW-Auswahl für eine HW-Implementierung gewählt wurden, sowie die aus diesen HW-Kandidaten resultierenden Datenpfade.

Die Knoten des Datenpfad-Ladegraphen repräsentieren die ausgewählten HW-Kandidaten. Sie enthalten weiterhin Daten über die aus den HW-Kandidaten entstehenden Datenpfade und die Häufigkeit ihrer Verwendung. Eine Kante zwischen zwei Knoten im Datenpfad-Ladegraph entsteht dann, wenn es einen Pfad im CFG gibt, welcher einen Ausgang des durch den ersten Knoten repräsentierten HW-Kandidaten und den Eingang des durch den zweiten Knoten repräsentierten HW-Kandidaten verbindet. Die Kanten enthalten Daten über eine abgeschätzte kürzeste Ausführungszeit von einem HW-Kandidaten zum nächsten. Ein Datenpfad-Ladegraph enthält genau einen Eingangsknoten und genau einen Ausgangsknoten.



Beispiel 6.2 Datenpfad-Ladegraph

Wenn beispielsweise in einer Prozedur drei HW-Kandidaten für die HW-Implementierung ausgewählt werden (Beispiel 6.2a), so bilden diese jeweils einen Knoten im Datenpfad-Ladegraph (DP1, DP2, DP3 in Beispiel 6.2b). Die zusätzlich im Datenpfad-Ladegraph enthaltenen Knoten DPB und DPE bezeichnen den Eintritt sowie den Austritt aus der Prozedur.



Beispiel 6.3 Globaler Datenpfad-Ladegraph

Diese Knoten werden benötigt, wenn es unter Umständen wie im Beispiel gezeigt vom Eintritts-Kontrollflussblock des CFG aus gesehen mehrere erreichbare HW-Kandidaten gibt. Außerdem kann es dementsprechend auch von mehreren HW-Kandidaten jeweils einen Pfad zum Ausgang des CFG geben. Durch das Einfügen der zusätzlichen Eingangs- und Ausgangsknoten wird also das Auftreten von mehr als einem Eingangsknoten und mehr als einem Aus-

gangsknoten verhindert. Besitzt eine Prozedur keine in HW realisierten HW-Kandidaten, so ist der Eingangsknoten direkt mit dem Ausgangsknoten verbunden. Daten wie der Ressourcenverbrauch der aus den HW-Kandidaten entstehenden Datenpfade sowie deren Ausführungsbeiwerte sind zusätzlich in den Knoten enthalten.

Die Beschränkung des Rekonfigurations-Schedulings auf den Datenpfad-Ladegraph einer einzelnen Prozedur kann aber die Anforderungen für ein gesamtes Programm nicht erfüllen. So bestehen Beziehungen auch zwischen den HW-Kandidaten unterschiedlicher Prozeduren. Außerdem kann eine Prozedur und damit die in ihr enthaltenen HW-Kandidaten in verschiedenen anderen Prozeduren Verwendung finden. Um dieser Anforderung gerecht zu werden, bietet sich die Erzeugung eines programmweiten Datenpfad-Ladegaphen aus denen der Prozeduren an. Dabei werden Informationen aus der Prozeduraufruf-Hierarchie verwendet, welche in den globalen (programmweiten) Datenpfad-Ladegraph einfließen (Beispiel 6.3).

Im Beispiel werden aus der `main`-Prozedur eines Programms die beiden Funktionen `g()` und `f()` aufgerufen, wobei `g()` wiederum `f()` aufruft. Um nun den globalen Datenpfad-Ladegraph (Beispiel 6.3c) aufzubauen, werden die den Funktionen zugehörigen Datenpfad-Graphen in den der Prozedur `main` eingefügt. Dabei werden die von den Eingangsknoten abgehenden und die zu Ausgangsknoten gehenden Kanten mit den Kanten im Datenpfad-Ladegraph der aufrufenden Prozedur verbunden. Das Aufrufen einer Prozedur an verschiedenen Positionen im Programm erzeugt keine zusätzlichen Knoten. Nur die Kanten werden in den globalen Graph eingefügt.

6.4 Heuristiken

Basierend auf dem Datenpfad-Ladegraph wurden für COMRADE verschiedene Heuristiken implementiert und erprobt, welche den Datenpfad-Ladegraph so partitionieren, dass aus jeder Partition eine Konfiguration erzeugt werden kann. Die Heuristiken werden in den folgenden Abschnitten beschrieben und mit experimentellen Ergebnissen bewertet. Der für die folgenden Beispiele zu Grunde liegende RL bietet 400 Ressourceneinheiten.

6.4.1 Höchstwahrscheinliche Verwendungspfade

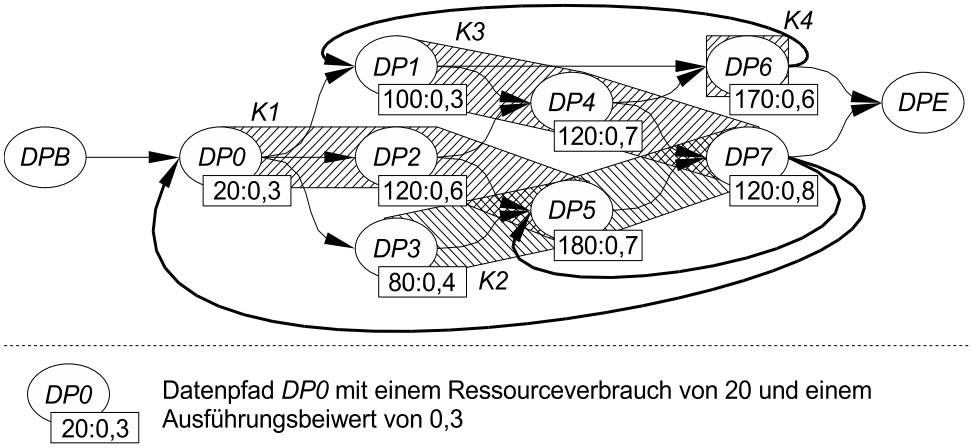
Bei dieser Heuristik beginnt der Algorithmus immer am Anfang eines Datenpfad-Ladegraphen. Er sucht Pfade in diesem, welche eine hohe Wahrscheinlichkeit der Ausführung haben. Dazu hat der Ausführungsbeiwert eines Datenpfads einen hohen Einfluss auf die Auswahl für eine bearbeitete Partition.

Ähnlich dem Algorithmus der Pfadselektion (Abschnitt 3.2.2) sucht sich diese Heuristik aus einer Arbeitsliste einen Anfangsknoten für eine neue Partition heraus. In dieser Liste befinden sich zu Anfang nur die Datenpfade, welche Nachfolger des Eingangsknotens sind. Weiterhin wird diese Liste bei der Zusammenstellung einer Konfiguration immer mit betrachteten, aber nicht zur Konfiguration hinzugefügten Datenpfaden aufgefüllt. Der Algorithmus endet, wenn die Arbeitsliste geleert wurde.

Beim Auffüllen einer Partition betrachtet der Algorithmus immer die Nachfolger des aktuellen Datenpfads. Er sucht sich aus diesen Nachfolgern den mit dem höchsten Ausführungsbeiwert aus. Kann dieser der aktuellen Partition hinzugefügt werden, ohne die zur Verfügung stehenden Ressourcen zu überschreiten, so wird er der Partition hinzugefügt. Alle nicht zur aktuellen Partition hinzugefügten Nachfolger des aktuellen Knoten werden der Liste angehängt.

Den Datenpfad-Ladegraph in Beispiel 6.4 beginnt der Algorithmus mit der Bearbeitung des Datenpfads *DP0*. Dieser wird in die erste Partition aufgenommen (*K1*). Aus den von diesem erreichbaren Nachfolgern wird nun *DP2* wegen des höchsten Ausführungsbeiwerts der aktuellen Partition hinzugefügt. Dasselbe geschieht mit *DP5*. Keiner der Nachfolger dieses Datenpfads kann nun mehr zu der aktuellen Partition hinzugefügt werden, ohne die Ressourcen zu überschreiten. Damit ist diese Partition vollständig bestimmt und es kann eine neue erzeugt werden.

In der Arbeitsliste ist nun als erster Eintrag der Datenpfad *DP3* enthalten, da dieser der erste betrachtete aber nicht verwendete Nachfolger-Datenpfad ist. Mit diesem sucht nun der Algorithmus eine weitere gültige Partition. Am Ende der Heuristik wurde der Datenpfad-Ladegraph in vier Partitionen aufgeteilt.

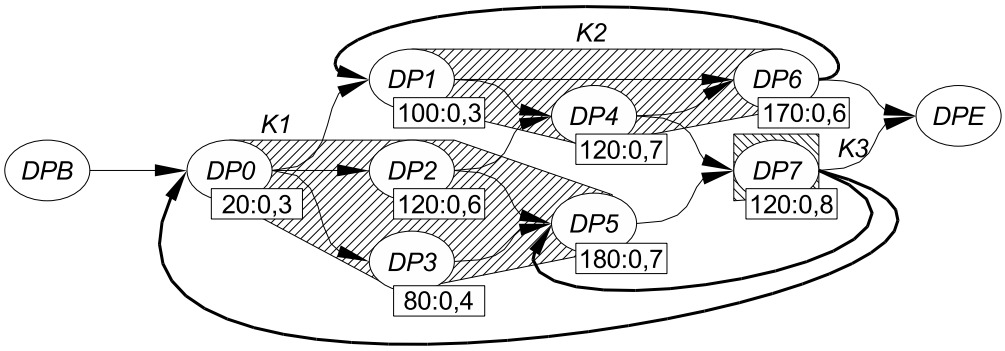


Beispiel 6.4 Partitionierung mittels Ausführungsbeiwert

Wie man leicht erkennen kann, ist diese Aufteilung für dieses Beispiel nicht ideal. Vor allem die Platzierung des Datenpfads *DP6* in einer eigenen Partition ist sehr ungünstig, da man annehmen kann, dass die sich in einer Schleife befindlichen Datenpfade *DP1*, *DP4* und *DP6* öfter nacheinander ausgeführt werden müssten. Außerdem stößt diese Heuristik an ihre Grenzen, wenn auf den Pfaden im Datenpfad-Ladegraph vom Eingangsknoten zum Endknoten nur sehr wenige Datenpfade enthalten sind. Hier kann es vorkommen, dass zwar hochwahrscheinliche Partitionen gebildet werden. In Einzelfällen nutzen sie aber die Ressourcen des Ziel-RLs nicht vollständig aus. Zum Umgehen dieser Nachteile wurden die in den nachfolgenden Abschnitten beschriebenen Heuristiken entwickelt.

6.4.2 Betrachten aller angrenzenden Datenpfade

Zur Lösung des Problems der Nichtbeachtung von benachbarten Datenpfaden soll durch eine Heuristik umgangen werden, welche die zu einer Partition hinzuzufügenden Datenpfade nicht nur aus den Nachfolgern des aktuell bearbeiteten Datenpfads heraussucht. Es werden vielmehr alle Datenpfade untersucht, welche durch Kanten mit mindestens einem Datenpfad der aktuellen Partition verbunden sind.



Beispiel 6.5 Partitionierung durch Betrachtung aller angrenzenden Datenpfade

Bei dieser Heuristik werden zwei verschiedene Strategien verfolgt. Wie auch in der vorigen werden neue Datenpfade der aktuellen Partition durch die Auswahl von Nachfolgern mit einem hohen Ausführungsbeiwert hinzugefügt. Nach jedem dieser Schritte betrachtet die Heuristik weiterhin alle durch Kanten mit der aktuellen Partition verbundenen Datenpfade, ob sich die Einbeziehung lohnen würde. Dabei wird ein Datenpfad in diesem Schritt hinzugenommen, wenn er mit mehr als zwei Kanten mit Datenpfaden der aktuellen Partition und mit nicht mehr Kanten mit Datenpfaden einer anderen Partition verbunden ist.

Als Ergebnis erzeugt die Heuristik eine Partitionierung, welche nicht nur einzelne Pfade, sondern auch Verzweigungsmöglichkeiten enthält, wenn diese lohnenswert erscheinen. In der Partitionierung des Beispielgraphen (Beispiel 6.5) ist erkennbar, dass sich gegenüber der vorigen Partitionierung eine Verbesserung dadurch ergibt, dass in Partition *K2* nun eine ganze Schleife enthalten ist. Leider können auch hier nicht beide Schleifen auf jeweils einer Partition zusammengefasst werden.

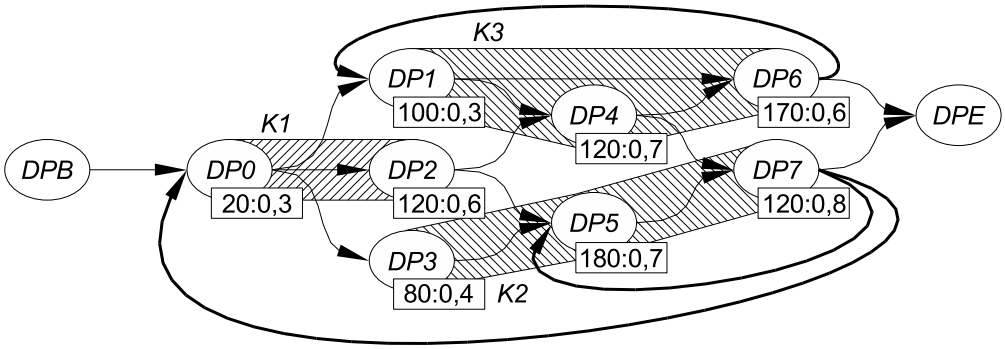
6.4.3 Zusammenfassen von Schleifen

Zur Lösung der Probleme durch das Aufbrechen von Schleifen werden diese bei einer weiteren Heuristik mit dem Zusammenfassen von Schleifen im Datenpfad-Ladegraph höher gewichtet. Bei diesem Ansatz werden deshalb im Datenpfad-Ladegraph zuerst Schleifen gesucht. Übersteigt die Summe der Ressourcen der Datenpfade in einer Schleife nicht die durch

den RL bereitgestellte Anzahl von Ressourcen, so bildet die Schleife eine Partition. Da meist aber nicht alle Datenpfade Mitglieder von Schleifen sind, müssen die übrigen Datenpfade noch zu Partitionen hinzugefügt oder geformt werden.

Dazu wird als erstes versucht, die verbliebenen Datenpfade den schon existierenden Partitionen hinzuzufügen. Alle durch Kanten mit einem Datenpfad in einer existieren Partition verbundenen Datenpfade werden dabei betrachtet und nach dem Ausführungsbeiwert gewichtet. Der Datenpfad, welcher zusammen mit der vorgegebenen Partition die auf dem RL verfügbaren Ressourcen nicht überschreitet und außerdem mit dieser Eigenschaft den größten Ausführungsbeiwert besitzt, wird der aktuellen Partition hinzugefügt. Dies geschieht so lange, bis kein Datenpfad mehr hinzugefügt werden kann.

Am Ende dieser Heuristik können immer noch Datenpfade nicht Partitionen zugeordnet worden sein. Auf diese werden dann die bereits zuvor beschriebenen Algorithmen angewandt.



Beispiel 6.6 Partitionierung mit Zusammenfassen von Schleifen

6.5 Praktische Ergebnisse

Um die Anwendbarkeit der Heuristiken zu testen, wurden diese auf die Benchmarks capacity, pegwit und versatility sowie ein Programm mit einer großen Menge an Schleifen (viele_fors) angewendet. Als Ziel-RLs wurden aus den in [Zufe05] veröffentlichten Daten die für die FPGAs Xilinx XC4013 (576CLBs), XC4036 (1296CLBs), XC4062 (2304CLBs) und Virtex XCV1000 (6144CLBs) ausgewählt. Dabei wurde angenommen, dass die zur Verfügung gestellten Ressourcen nur zu 80% ausgenutzt werden.

In den experimentell ermittelten Daten (Tabelle 6.7) werden zu dem jeweiligen Ziel-RL in der zweiten Spalte die Anzahl der benötigten Rekonfigurationen (Rek.) und die Anzahl der Gesamtressourcen (Res.) gegeben. Die Gesamtressourcen ergeben sich aus der Summe der Ressourcen aller Konfigurationen, die nacheinander auf den RL geladen werden müssten. Die dort auftretenden Zahlen können je nach Ziel-RL für den gleichen Benchmark variieren, obwohl keine Auswahl zwischen HW-SW-Ausführung stattfindet, sondern immer die HW-Version verwendet wird. Der Grund für die Variationen ist darin zu sehen, dass die Pfadselek-

tion die aus gleichen Schleifen entstehenden Datenpfade je nach zur Verfügung gestellter Ressourcenanzahl vergrößert oder verkleinert. Ist die vom Ziel-RL zur Verfügung gestellte Anzahl von Ressourcen groß, dann können bei der Pfadselektion mehr Pfade in die HW-Kandidaten aufgenommen werden. Die HW-Kandidaten verwenden somit auch in der Summe mehr Ressourcen.

Damit ändert sich auch die Größe der rekonfigurierten Gesamtfläche (Res.). Die Änderungen in der Anzahl der Rekonfigurationen bei pegwit liegen darin begründet, dass HW-Kandidaten teilweise durch auftretende Komplikationen bei der Übersetzung in einen Datenpfad doch nicht für eine HW-Ausführung vorgesehen sind. An den Zahlen ist aber zu erkennen, dass diese Anzahl nicht sehr groß ist.

Ziel-RL	Ohne Partit. Rek./Res.	Heuristik Wahrsch. %Rek./%Res.	Heuristik Nachbarn %Rek./%Res.	Heuristik Schleifen %Rek/%Res.	Theorith. Optimum %Rek.
capacity					
XC4013	42/1344	2,4/16,6	2,4/16,6	2,4/16,6	2,4
XC4036	42/8512	4,8/13,2	4,8/13,2	9,5/26,3	4,8
XC4062	42/9408	4,8/21,4	4,8/21,4	4,8/21,4	4,8
XCV1000	42/5408	2,7/23,7	2,7/23,7	2,7/23,7	2,4
pegwit					
XC4013	134414/21213976	62,6/174,2	64,0/139,0	64,9/149,9	-
XC4036	134008/21850512	0,9/5,0	64,0/217,2	3,8/16,6	-
XC4062	130342/57812812	62,6/191,7	64,0/211,9	0,9/3,6	-
XCV1000	129536/50191733	63,5/751,7	64,0/480,9	3,6/38,6	-
versatility					
XC4013	5380/172368	0,02/0,3	0,07/0,3	0,02/0,3	0,02
XC4036	5381/173824	0,1/1,4	0,1/1,4	0,1/1,0	0,06
XC4062	5381/175634	0,1/2,4	0,1/2,1	0,1/3,8	0,06
XCV1000	5381/10427290	66,6/133,0	66,6/100,0	66,6/133,6	66,6
viele_fors					
XC4013	12668/1503520	30,5/98,5	31,8/102,8	44,3/116,2	-
XC4036	12668/4293664	41,9/113,4	40,8/100,6	42,7/103,2	-
XC4062	12668/4390768	24,6/128,2	24,6/98,6	23,5/94,2	-
XCV1000	12668/2849088	0,04/0,8	1,9/14,9	1,9/15,1	-

Tabelle 6.7 Eingesparte Rekonfigurationen/Ressourcen

Weiterhin ist festzustellen, dass die Funktion der Pfadauswahl gewährleistet ist. Mit steigender von dem Ziel-RL zur Verfügung gestellter Ressourcenanzahl wächst auch die

Anzahl der zu rekonfigurierenden Gesamtressourcen und damit auch der durchschnittliche Ressourcenverbrauch der einzelnen Datenpfade. Nur bei der Verwendung eines XCV1000 anstatt des mit weniger CLBs ausgestatteten XC4062 scheint diese Regel nicht zu stimmen. Diese Ergebnisse kommen daher, dass erstens die CLBs für die XC4000-FPGAs und die Virtex-FPGAs und zweitens auch die Modulgeneratoren in GLACE unterschiedlich sind.

In den Spalten drei bis fünf wird der prozentuale Zuwachs an verwendeten Rekonfigurationen und Gesamtressourcen für die unterschiedlichen Heuristiken prozentual gegenüber den in Spalte zwei zu findenden Werten aufgetragen. Diese zeigen, dass die Anzahl der Rekonfigurationen je nach Benchmark durch das Zusammenfassen von Datenpfaden zu Konfigurationen beträchtlich reduziert werden kann (Bei viele_fors bis zum Faktor 5000).

Betrachtet man beispielsweise die Ergebnisse der Heuristiken für capacity und das FPGA XC4013, dann würde man während eines Programmlaufs 42-mal rekonfigurieren müssen, wenn man keine Datenpfade zusammenfasst. Während des Programmlaufs würden insgesamt 1344 CLBs konfiguriert werden müssen. Fasst man nun die Datenpfade durch die Heuristiken zu Konfigurationen zusammen, muss man nur noch 2,4% der 42-mal (1-mal) rekonfigurieren müssen und würde damit insgesamt nur16,6% von 1344 CLBs rekonfigurieren müssen. In der letzten Spalte kann man erkennen, dass die 2,4% der Konfigurationen auch das optimale Ergebnis ist.

Wenn man jedoch annimmt, dass die insgesamt für eine Rekonfiguration benötigte Zeit linear mit der Anzahl der zu rekonfigurierenden Ressourcen wächst, so ist das Ergebnis bei den Heuristiken nicht mehr so gut. Nun muss man die zu konfigurierenden Gesamtressourcen betrachten. Diese können beim Zusammenfassen zu Konfigurationen sogar um ein mehrfaches ansteigen (751% bei pegwit). Für COMRADE hat dies aber keine Bedeutung, da die HW-Module in den HW-Realisierungen der Datenpfade nur mit großem Aufwand so platziert werden können, dass sich die Rekonfigurationszeit auch wirklich linear zu den verbrauchten Ressourcen verhält.

Weiterhin kann man aus den Daten entnehmen, dass sich nicht alle Heuristiken gut für die Reduzierung der Rekonfigurationsanzahl eignen. Vor allem die Heuristik, welche Schleifen für die Platzierung auf Konfigurationen priorisiert, erzielt bei allen Benchmarks gute bis sehr gute Ergebnisse.

In der letzten Spalte wurden für einige Benchmarks die optimalen Konfigurationen in Bezug auf die Minimierung der für einen Programmlauf aufzuwendenden Rekonfigurationen ermittelt. Dafür wurde genau ermittelt, zu welchem Zeitpunkt welcher Datenpfad geladen werden muss (Loop-Traces). Im Datenpfad-Ladegraph wurden die Datenpfade zu allen möglichen Konfigurationen zusammengefasst. Mit Hilfe der Loop-Traces konnte nun festgestellt werden, welche Aufteilung in Konfigurationen eine minimale Anzahl von Rekonfigurationen ergibt. An den Daten kann man erkennen, dass die von den Heuristiken erzeugten Lösungen meist denen des Optimums gleichen oder nur wenig abweichen.

6.6 Erweiterungsmöglichkeiten

Die Testdaten für das Rekonfigurations-Scheduling zeigen, dass die hier angegebenen, einfachen Heuristiken auch auf der Grundlage weniger Ausgangsdaten in der Lage sind, gute Ergebnisse zu erzielen. Die recht aufwendig zu erstellenden Loop-Trace-Daten sind hierfür nicht notwendig.

Einschränkend muss aber erwähnt werden, dass die Heuristiken bisher nur an einer kleinen Anzahl von Programmen getestet wurden. Wichtig wäre es, diese auf mehr Testprogramme anzuwenden und die so erzielten Ergebnisse wieder in die Heuristiken einfließen zu lassen (CE_11).

Bei den bisherigen Heuristiken wurde weiterhin als Optimierungsziel vor allem die Anzahl der während eines Programmlaufs zu erfolgenden Rekonfigurationen gesehen. Da die Rekonfigurationszeit eines FPGA oft von der Größe der Konfigurationsdatei und damit von der Anzahl der zu konfigurierenden Ressourcen linear abhängt, wäre auch eine andere Strategie wünschenswert. So hat sich herausgesellt, dass man andere Ergebnisse erhält, wenn man als Optimierungsziel die Gesamtanzahl der zu konfigurierenden Ressourcen ansehen würde. Mit diesem Optimierungsziel könnten auch Heuristiken gute Ergebnisse liefern, die nicht versuchen, die Ressourcen des RLs bis zum maximalen Wert ausnutzen. Auch einzelne Datenpfade könnten so eine Partition bilden und trotzdem die Summe der insgesamt zu rekonfigurierenden Ressourcen minimieren (CE_12).

Bisher außerdem unberücksichtigt geblieben ist der Einfluss des Rekonfigurations-Schedulings auf die HW-SW-Auswahl. Für die HW-SW-Auswahl wird bisher angenommen, dass die Rekonfigurationszeiten immer mit zur Laufzeit des HW-Kandidaten hinzugerechnet werden. Durch das Rekonfigurations-Scheduling müssen nun aber diese Zeiten nicht mehr für jeden HW-Kandidaten berücksichtigt werden. Eine mögliche Strategie zur Lösung dieses Problems wäre es, bei der HW-SW-Auswahl die Rekonfigurationszeit nicht zu beachten. Das Rekonfigurations-Scheduling müsste dann aber in die Partitionierung immer die Rekonfigurationszeit mit einbinden (CE_13). Die eigentliche HW-SW-Auswahl würde so also das Rekonfigurations-Scheduling übernehmen.

Kapitel 7

Zusammenfassung und Ausblick

In einem Adaptiven Computersystem werden einem Standardprozessor rechenintensive Aufgaben durch einen rekonfigurierbaren Logikbaustein abgenommen. Es handelt sich also um eine individuell anpassbare Spezial-Hardware. Dazu müssen Teile eines Programms als Spezialschaltung für einen rekonfigurierbaren Logikbaustein entwickelt werden. Diese Hardware-Schaltung rechnet dann viel schneller und vor allem mit dramatisch weniger Verlustleistung.

Die manuelle Programmentwicklung für ein Adaptives Computersystem verlangt vom Entwickler neben Software-Kenntnissen vor allem auch fundiertes Wissen in der Modellierung von Hardware. Bei der Kopplung des Standardprozessors mit dem rekonfigurierbaren Logikbaustein müssen Datenaustausch, konkurrierender Zugriff auf den Arbeitsspeicher und andere Probleme gelöst werden. Fachkräfte mit diesen Eigenschaften sind rar und teuer. Die manuelle Aufteilung in Hardware und Software sowie die getrennte Modellierung beider Teile erfordert einen für viele Projekte untragbaren Zeitaufwand.

Daher ist der Einsatz von Compilern für Adaptive Computersysteme von allergrößtem Interesse, welche aus einer Hochsprache automatisch Teile für die Hardware-Beschleunigung aussuchen, diese als Hardware synthetisieren und Schnittstellen für die Kommunikation zwischen Hardware und Software integrieren. Bestehende Compiler für Adaptive Computersysteme sind nur für ganz bestimmte Zielarchitekturen geeignet und können nur Subsets der Hochsprache C bearbeiten. Deshalb wurde in dieser Arbeit der Forschungs-C-Compiler COMRADE für Adaptive Computersysteme entwickelt. Diverse Verbesserungen gegenüber bisherigen Systemen wurden durch Tests mit realen Benchmarks belegt.

COMRADE nutzt eine Modulbibliothek für Hardware-Komponenten und Konfigurationsdateien und kann so leicht an neue Zielarchitekturen angepasst werden. Stehen für die adaptive Hardware Bibliotheksmodule bereit, kann eine neue Zielarchitektur nur durch eine geänderte Konfigurationsdatei ohne Veränderung von COMRADE selbst unterstützt werden.

Ein großer Vorteil ist die Unterstützung des vollen C-Sprachumfangs: COMRADE kann jeden C-Quelltext bearbeiten. Durch eine verbesserte Hardware-Software-Partitionierung kann der synthetisierte Hardware-Anteil vergrößert werden. Durch eine Reduktion der Zahl der Rekonfigurationen während der Programmausführung ergibt sich weiteres Beschleunigungspotential.

Die Rekonfigurationszeit hat sich während des Projekts als behindernder Faktor für eine Hardware-Beschleunigung herausgestellt. Außerdem sind die durch COMRADE synthetisierten Hardware-Teile verhältnismäßig klein gegenüber den auf aktuellen rekonfigurierbaren Logikbausteinen verfügbaren Ressourcen. Deswegen wurde zur Vermeidung ständigen Re-

konfigurierens die Zusammenfassung mehrerer Hardware-Teile zu einer Konfiguration untersucht. Die hierbei entwickelten Heuristiken können schon mit wenigen gegebenen Daten Konfigurationen erzeugen, die während eines Programmlaufs bis zu 99,98% der Konfigurationen einsparen. Die Ergebnisse sind damit nur unwesentlich schlechter als die optimaler Konfigurationen.

Für die Steuerung der Hardware wurde ein neuer Kontroller mit Vorteilen gegenüber bisherigen Datenflussmaschinen entwickelt. Neben Hardware-Pipelines unterstützt der Kontroller auch die spekulative Berechnung von Ergebnissen, aus denen anschließend das richtige für die weitere Berechnung ausgewählt wird. Durch ein neuartiges auf Petri-Netzen aufbauendes Ausführungsmodell können spekulativ begonnene Berechnungen, deren Ergebnisse nicht mehr benötigt werden, frühzeitig abgebrochen werden. Gegenüber herkömmlichen spekulativen Mechanismen ergibt sich ein Laufzeitgewinn, da nicht mehr auf alle spekulativen Ergebnisse gewartet werden muss. Ein weiterer Vorteil des Kontrollers ist die Behandlung unterschiedlicher Typen von Hardware-Operatoren. Diese können eine feste wie auch eine variable Laufzeit haben und werden trotzdem optimal integriert.

Die Anwendung verschiedener Optimierungen auf Hochsprachenebene hat gezeigt, dass diese wesentlichen Einfluss auf die Größe der erzeugten Hardware besitzen. So konnte nachgewiesen werden, dass einerseits der Einfluss jeder einzelnen Optimierung Vorteile erbringt, die Kombination von Optimierungen aber den Effekt noch erhöht. Durch die in COMRADE implementierten Optimierungen wie Constant-Propagation, Bitbreitenreduktion und skalare Ersetzung konnten der Hardware-Verbrauch der Operatoren wie auch die Anzahl der Speicherzugriffe wesentlich reduziert werden. Ressourcenverbrauch und Geschwindigkeit der Hardware werden so positiv beeinflusst.

Die Arbeit mit COMRADE hat gezeigt, dass die vorgestellten Algorithmen Verbesserungen gegenüber bestehenden anderen Compilersystemen erbringen. Dennoch brachten jede Implementierung und jeder Test wiederum neue Ideen hervor, welche für die zukünftige Entwicklung von COMRADE interessant sein können. Details sind jedem Kapitel dieser Arbeit als Abschnitt *Erweiterungsmöglichkeiten* hinzugefügt. Vor allem von den vorgeschlagenen Möglichkeiten zur Vereinheitlichung von Hardware-Operatoren würde die Handhabbarkeit von COMRADE profitieren. Algorithmen würden vereinfacht und übersichtlicher. Unterschiedliche Typen müssen nicht mehr unterschiedlich behandelt werden. Aber auch die Verfeinerung der Abschätzungen in COMRADE kann die künftige Anwendbarkeit erhöhen.

Die durch einen Compiler erzielbaren Ergebnisse sind stark mit den zugrunde liegenden Architekturen verbunden. Wenn ein Prozessor eine bestimmte Abfolge von Befehlen nur mangelhaft ausführen kann, muss diese vom Compiler vermieden werden und kann so unter Umständen nicht mehr höchstperformante Programme erzeugen. So stellt für COMRADE vor allem die lange Rekonfigurationszeit ein Problem dar, die Anpassung der Hardware während eines Programmlaufs verbraucht bei kleinen Programmen oft schon soviel Zeit wie das Programm allein auf dem Prozessor. Die Hardware-Beschleunigung wird somit stark ausgebremst.

Da die Rekonfigurationszeit direkt von der Flexibilität der rekonfigurierbaren Hardware abhängt, wäre bei neuen Architekturen eine Rekonfigurations-Reduktion wünschenswert. Bei der Übersetzung aus C-Quelltext ist diese sogar möglich, da nur eine begrenzte Menge an Operationen benötigt wird und darüber hinaus die verwendeten Bitbreiten meist ganz oder fast ganz auch mit Operatoren mit beispielsweise vier Bit Breite realisiert werden könnten. Würde man nur die Bitbreitenerhöhung der Operatoren gegenüber denen von herkömmlichen FPGAs mit einem Bit Breite betrachten, so könnten schon drei Viertel des Rekonfigurations-Aufwands eingespart werden. Die Einschränkung der Operationsvielfalt verspricht aber noch mehr Einsparungspotential.

Verbesserte Adaptive Computersysteme und angepasste Compiler könnten künftig den Drang nach immer schnelleren Prozessoren dämpfen. Adaptive Computer könnten durch verringerten Strombedarf erheblich günstiger mit Wärme und Energie umgehen. Der Weg zu immer effizienteren Adaptiven Computersystemen ist zwar noch längst nicht zu Ende, verspricht aber immer komfortabler zu werden.

Literatur

Verweise auf Druckerzeugnisse

Alle nachfolgenden Verweise in diesem Abschnitt wurden durch Verlage oder als Arbeiten an Universitäten bzw. in Firmen in gedruckter Form veröffentlicht:

[Acka00] Ackad, C., *Optimierte automatische Statechart-Implementierungen im Software- und Hardware-Entwurf eingebetteter Systeme*, Dissertation, Technische Universität Braunschweig, 2000

[AhBT04] Ahmadinia, A., Bobda, C., Teich, J., *On-line Placement for Dynamic Reconfigurable Devices*, International Journal of Embedded Systems (IJES), 2004

[AJLA95] Allan, V. H., Jones, R. B., Lee, R. M., Allan, S: J., *Software Pipelining*, ACM Computing Surveys, volume 27, number 3, 1995

[AkJa02] Akturan, C., Jacome, M. F., *RS-FDRA: a register sensitive software pipelining algorithm for embedded VLIW processors*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems volume 21 number 12, December 2002

[AlKe02] Allen, R., Kennedy, K., *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann Publishers, 2002

[Appe98] Appel, A., *Modern Compiler Implementation in C*, Cambrigde University Press, 1998

[BaGS93] Bacon, D. F., Graham, S. L., Sharp, O. J., *Compiler Transformations for HighPerformance Computing*, Computer Science Division Technical Report UCB--CSD-- 93--781, U. C. Berkeley, 1993

[BaLa93] Ball, T., Larus, J. R., *Branch Prediction for free*, Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, 1993

[BaZo93] Barret, D. A., Zorn, B. G., *Using lifetime predictors to improve memory allocation performance*, In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Albuquerque, 1993

[BeVo03] Becker, J., Vorbach, M., *PACT XPP Architecture in Adaptive System-on-Chip Integration*, Proc of ERSA'03, Las Vegas, 2003

[BoDP98] Borrione, D., Dushina, J., Pierre, L., *Formalization of Finite State Machines with Data Path for the Verification of High-Level Synthesis*, In Proceedings of the IEEE Brazilian Symposium on Integrated Circuit Design, 1998

[BrCo94] Briggs, P., Cooper, K. D., *Effective partial redundancy elimination*, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 1994

[Buck93] Buck, J. T., *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*, Doktorarbeit University od California Berkeley, 1993

[BuGo02] Budiu, M., Goldstein, S. C., *Pegasus: An Efficient Intermediate Representation*, CMU CS Technical Report, 2002

[BuGo03] Budiu, M., Goldstein, S., *Optimizing memory accesses for spatial computation*, In Proceedings of the 1st International ACM/IEEE Symposium on Code Generation and Optimization (CGO 03), 2003

[CaHW00] Callahan, T., Hauser, R., Wawrzynek, J., *The GARP architecture and C Compiler*, IEEE Computer 33(4), April 2000

[CaEA99] Carter, L., Simon, B., Calder, B., Carter, L., Ferrante, J., *Predicated Static Single Assignment*, In Proceedings of International Conference on Parallel Architectures and Compilation Techniques, 1999

[CaCK90] Callahan, D., Carr, S., Kennedy, K., *Improving register allocation for subscripted variables*, In Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, 1990.

[Call02] Callahan, T., *Automatic Compilation of C for Hybrid Reconfigurable Architectures*, PhD thesis, University of California at Berkeley, 2002

[CaWa98] Callahan, T., Wawrzynek, J., *Instruction Level Parallelism for Reconfigurable Computing*, 8th International Workshop on Field-Programmable Logic and Applications, Estonia, 1998

[ChMH91] Chang, P. P., Mahlke, S. A., Hwu, W. W., *Using profile information to assist classic code optimizations*, Software-Practice and Experience 21, 1991

[CMCH92] Chang, P. P., Mahlke, S. A., Chen, W. Y., Hwu, W. W., *Profile-guided automatic inline expansion for C programms*, Software-Practice and Experience 22, 1992

[DaPR01] Dandalis, A., Prasanna, V. K., Rolim, J. D. P., *A Comparative Study of Performance of AES Final Candidates Using FPGAs*, Lecture Notes in Computer Science, Volume 1965, 2001

[DeCH98] Deitrich, B. L., Cheng, B., Hwu, W. W., *Improving Static Branch Prediction in a Compiler*, Proceedings of PACT'98, 1998

[DiEA01] Diniz, P., Hall, M., Park, J., So, B. Ziegler, H., *Bridging the Gap between Complation and Synthesis in the DEFACTO System*,. Proceedings of the 14th Workshop on Languages and Compilers for Parallel Computing Synthesis (LCPC'01), 2001

[Erns97] Ernst, R., *Hardware/Software Co-Design of Embedded Systems*, Asia Pacific Conference on Computer Hardware Description Languages, Taiwan, 1997

[FeKT01] Fekete, S., Köhler, E., Teich, J., *Optimal FPGA Module Placement with Temporal Precedence Constraints*,.In Proc. of DATE 2001 (Design, Automation and Test in Europe), Computer Society Press, 2001

[FiKS00] Fink, S., Knobe, K., Sarkar, V., *Unified Analysis of Array and Object References in Strongly Typed Languages*, In Seventh International Static Analysis Symposium, Juni 2000

[FoFA00] Foster, J. S., Fahndrich, M., Aiken, A., *Polymorphic versus Monomorphic Flow-Insensitive Points-to Analysis for C*, Static Analysis Symposium, 2000

[Golz04] Golze, U., *Einführung in den VLSI-Entwurf*, Vorlesungsskript Technische Universität Braunschweig, 2004

[GrKM82] Graham, S. L., Kessler, P. B., McKusick, M. K., *gprof: a Call Graph Execution Profiler*, SIGPLAN Symposium on Compiler Construction, 1982

[HaLe97] Ha, S., Lee, E. A., *Compile-time scheduling of dynamic constructs in dataflow program graphs*, IEEE Transactions on Computers 46(7), July 1997

[HeEr98] Henkel, J., Ernst, R., *High-Level Estimation Techniques for Usage in Hardware/Software Co-Design*, Asia and South Pacific Design Automation Conference, Japan, 1998

[HiPo01] Hind, M., Pioli, A., *Evaluating the effectiveness of pointer alias analyses*, Science of Computer Programming, 2001

[HMBG93] Hank, R. E., Mahlke, S. A., Bringmann, R. A., Gyllehaal, J. C., Hwu, W. W., *Superblock Formation Using Static Program Analysis*, Micro-26, 1993

[HoEr95] Holtmann, U., Ernst, R., *Combining MBP-speculative computation and loop pipelining in high-level synthesis*, Proceedings of the 1995 European Conference on Design and Test, 1995

[HTKZ04] Hecht, R., Timmermann, D., Kubisch, S., Zeeb, E., *Network-on-Chip basierende Laufzeitsysteme für dynamische rekonfigurierbare Hardware*. International Conference on Architecture of Computing Systems (ARCS), 2004

[JuHw94] Jun, H. S., Hwang, S. J., *Automatic synthesis of pipeline structures with variable data initiation intervals*, Proceedings of the 31st annual Conference on Design Automation, 1994

[Kahn74] Kahn, G., *The Semantics of a Simple Language for Parallel Programming*, In Proceedings of IFIP congress, 1974

[KaKo04] Kasprzyk, N., Koch, A., *Verbesserte Hardware-Software-Partitionierung für Adaptive Computer*, International Conference on Architectures of Computing Systems, Augsburg, 2004

[Kasp05] Kasprzyk, N., COMRADE: Current implementation and future, Technischer Report 2005-01, Abteilung Entwurf integrierter Schaltungen, Technische Universität Braunschweig, 2005

[KiRD00] Kienhuis, B., Rijpkema, E., Deprettere, E., *Compaan: Deriving process networks from matlab for embedded signal processing architectures*, In 8th International Workshop on Hardware/Software Codesign, 2000

[KKGR03] Kasprzyk, N., Koch, A., Golze, U., Rock, M., *An Improved Intermediate Representation for Datapath Generation*, Conference on Engineering of Reconfigurable Systems and Algorithms, 2003

[KnSa98] Knobe, K., Sarkar, V., *Array SSA form and its use in Parallelization*, In 25th Annual ACM SIGACTSIGPLAN Symposium on the Principles of Programming Languages, January 1998

[Koch99] Koch, A., *Enabling Automatic Module Generation for FCCM Compilers*, Proc. IEEE Symp. On Field-Programmable Custom Computing Machines, 1999

[Koch00] Koch, A., *A Comprehensive Prototyping Platform for Hardware-Software Codesign*, Workshop on Rapid Systems Prototyping, Paris, 2000

[KoKa02] Koch, A., Kasprzyk, N., *Module Generator-based Compilation for Hardware-Software Codesign*, IEEE Intl. Symp. On FCCMs, 2002

[KoWo02] Kountouris, A. A., Wolinski, C., *Efficient scheduling of conditional behaviors for high-level synthesis*, ACM Transactions on Design Automation of Electronic Systems, Volume 7 , Issue 3, Juli 2002

[Krah03] Krahn, H., *Baumhöhenreduktion zur Laufzeitoptimierung von adaptiven Rechensystemen*, Studienarbeit an der Abteilung Entwurf integrierter Schaltungen (E.I.S.), Technische Universität Braunschweig, 2003

[LaRJ98] Lakshminarayana, G., Raghunathan, A., Jha, N. K., *Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions*, Proceedings of the 35th annual conference on Design automation, 1998

[LeEA01] Leong, P., Leong, M., Cheung, O., Tung, T., Kwok, C., Wong, M., Lee, K., *Pilchard - a reconfigurable computing platform with memory slot interface*, In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, April 2001

[LeeA03] Lee, E., *Overview of the Ptolemy Project*, Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, USA, 2003

[LiEA00] Li, Y., Callahan, T., Darnell, V., Harr, R. , Kurkure, U., Stockwood, J., *Hardware-Software Co-Design of Embedded Reconfigurable Architectures*, In Proceedings of the 37th Conference on Design Automation, 2000

[LiLa97] Lim, A. W., Lam, M. S., *Maximizing Parallelism and Minimizing Synchronization with Affine Transforms*, Proceedings of the Twenty-fourth Annual {ACM} Symposium on the Principles of Programming Languages, 1997

[LGDR03] Lavenier, D., Guyetant, S., Derrien, S., Rubini, S., *A Reconfigurable Parallel Disk System for Filtering Genomic Banks*, Engineering of Reconfigurable Systems and Algorithms 2003

[MAGH94] Wagner, T. A., Maverick, V., Graham, S. L., Harrison, M. A., *Accurate static estimators for program optimization*, In Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, 1994.

[MaHo00] Maruyama, T., Hoshino, T., *A C to HDL Compiler for Pipeline Processing on FPGAs*, IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2000

[MaMG99] Manne, S., Mudge, T., Grunwald, D., *Cool Chips Tutorial*, 32nd Annual International Symposium on Microarchitecture (MICRO), 1999

[MaRi01] Marinescu, M-C. V., Rinard, M., *High-level automatic pipelining for sequential circuits*, Proceedings of the 14th International Symposium on Systems Synthesis, 2001

[MeEA03] Mei, B., Vernalde, S., Verkest, D., De Man, H., Lauwereins, R., *Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling*, Design, Automation and Test in Europe Conference and Exhibition (DATE'03)

[Moll04] Moll, M., *Kontrollflussabhängige Analyse von Feldzugriffe mit Hilfe der Omega-Bibliothek*, Diplomarbeit an der Abteilung Entwurf integrierter Schaltungen (E.I.S.), Technische Universität Braunschweig, 2004

[Much97] Muchnik, S. S., *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, San Francisco, California, 1997

[Mück03] Mücke, S., *Bitweise Optimierung – Compile-Zeit-Auswertung von Konstanten zur Reduktion von Bitbreiten*, Studienarbeit an der Abteilung Entwurf integrierter Schaltungen (E.I.S), Technische Universität Braunschweig, 2003

[NoEA03] Nollet, V., Mignolet, J., Bartic, A., Verkest, D., Vernalde, S., Lauwereins, R., *Hierarchical Run-Time Reconfiguration Managed by an Operating System for Reconfigurable Systems*, Engineering of Reconfigurable Systems and Algorithms, 2003

[Patt95] Patterson, J. R. C., *Accurate Static Branch Prediction by Value Range Propagation*, ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, 1995

[Pete81] Peterson, J. L., *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Englewood Cliffs, 1981

[Petk01] Petkov, D., *Efficient Pipelining of Nested Loops: Unrolland –Squash*, Master of Engeneering thesis, Massachusetts Institute of Technology, January 2001

[Poll99] Pollak, F., *New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies*, 32nd Annual International Symposium on Microarchitecture (MICRO), 1999

[Pugh92] Pugh, W., *Uniform techniques for loop optimization*, Proceedings of International Conference on Supercomputing, 1992

[RaRL00] Raghunathan, V., Ravi, S., Lakshminarayana, G., *High-Level Synthesis with Variable-Latency Components*, Proceedings of the 13th International Conference on VLSI Design, 2000

[ShMo98] Shim, S., Moon, S-M., *Split-path Enhanced Pipeline Scheduling for Loops with Control Flows*, International Symposium on Microarchitecture, 1998

[SSHW94] Stoltz, E., Srinivasan, H., Hook, J., Wolfe, M., *Static single assignment form for explicitly parallel programs: Theory and practice*, Tech. report, Dept. of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, Portland, Oregon, 1994.

[StBA00] Stephenson, M., Babb, J., Amarasinghe, S., *Bitwidth analysis with application to silicon compilation*, In Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation, 2000

[Stee96] Steensgaard, B., *Points-to Analysis in Almost Linear Time*, Symposium on Principles of Programming Languages, 1996

[StLu03] Styles, H., Luk, W., *Branch Optimisation Techniques for Hardware Compilation*, 13th International Conference on Field Programmable Logic and Application, 2003

[TeFS99] Teich, M., Fekete, S., Schepers, J., *Compile-Time Optimization of Dynamic Hardware Reconfigurations*, Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), 1999

[TuPa95] Tu, P., Padua, D. A., *Gated SSA-based Demand-Driven Symbolic Analysis for Parallelizing Compilers*, International Conference on Supercomputing, 1995

[UHGB04] Ullmann, M., Hübner, M., Grimm, B., Becker, J., *An FPGA Run-Time System for Dynamical On-Demand Reconfiguration*, Reconfigurable Archi-tectures Workshop (RAW), 2004.

[ViEA02] Villarreal, J., Suresh, D., Stitt, G., Vahid, F., Najjar, W., *Improving Software Performance with Configurable Logic*, Design Automation for Embedded Systems, 7, 325-339, Kluver Academic Publishers, 2002

[Wein01] Weinhardt, M., Luk, W., *Pipeline Vectorization*, In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Feb. 2001

[WeLu01] Weinhardt, M., Luk, W., *Task-Parallel Programming of Reconfigurable Systems*, Lecture Notes in Computer Science, volume 2147, 2001

[WeZa91] Wegman, M. N., Zadeck, F. K., *Constant Propagation with Conditional Branches*, ACM Transactions on Programming Languages and Systems, Vol. 13, No. 2, 1991

[WiKe01] Wigley, G., Kearney, D., *The Development of an Operating System for Reconfigurable Computing,* In Proc. IEEE Symp. FCCM'01, April 2001, IEEE Press.

[Wint03] Winter, G., *Implementierung SSA-basierter Optimierungen für adaptive Rechensysteme*, Studienarbeit an der Abteilung Entwurf integrierter Schaltungen (E.I.S.), Technische Universität Braunschweig, 2003

[WuLa94] Wu, Y., Larus, J. R., *Static Branch Frequency and Program Profile Analysis*, 27th IEEE/ACM Symposium on Microarchitectures (MICRO-27), 1994

[Xili96] Xilinx, *XC6200 field programmable gate arrays*, Technical report, Xilinx, Inc., 1996

[YeSB00] Ye, A., Shenoy, N., Banerjee, P., *A C compiler for a processor with a reconfigurable functional unit,* ACM International Symposium on Field Programmable Gate Arrays, 2000

[YoHR99] Yong, S. H., Horwitz, S., Reps, T. W., *Pointer Analysis for Programs with Structures and Casting*, SIGPLAN Conference on Programming Language Design and Implementation, 1999

[Zufe05] Zufelde, K., *Rekonfigurations-Scheduling von Datenpfaden für Adaptive Computersysteme*, Diplomarbeit an der Abteilung Entwurf integrierter Schaltungen (E.I.S.), Technische Universität Braunschweig, 2005

Verweise auf Internetquellen

Die folgenden Verweise sind Verweise auf Quellen im Internet zurzeit der Erstellung der Doktorarbeit. Die Seiten dieser Verweise werden auf der beiliegenden CD im Unterverzeichnis *Arbeit→Referenzen→Webseiten* zur Verfügung gestellt.

[Celo03] Celoxica Limited, *Handel-C Language Reference Manual*, http://www.celoxica.com/techlib/files/CEL-W0410251JJ4-60.pdf

[Elix01] Elixent, *Changing the electronic landscape*, White paper, http://www.elixent.com/assents/WP0001_D_FABrix_Apps.pdf

[Haus97] Hauser, J. R., *The Garp architecture,* http://brass.cs.berkeley.edu/documents/GarpArchitecture.ps

[Open04] Open64 compiler, http://sourceforge.net/projects/open64

[Xili03] Xilinx, *Virtex-II Pro™ Platform FPGAs: Complete Data Sheet*, http://www.xilinx.com

Index

Abkürzungen

AST	Abstrakter Syntaxbaum
CE	COMRADE-Erweiterung
CFG	Kontrollflussgraph (Control flow graph)
CLK	Rekonfigurierbarer Logikblock
COMRADE	Compiler für Adaptive Computersysteme
DFG	Datenflussgraph
DFCSSA	Dataflow-Controlled SSA
DSP	Digital Signal Processor
ECFG	Kanten-Kontrollflussgraph
EPS	Embedded Processor Subsystem
FPGA	Field-Programmable Gate-Array
HW	Hardware
IP	Intellectual Property
PAE	Processing Array Unit
RAP	Reconfigurable Algorithm Processor
RL	Rekonfigurierbarer Logikbaustein
RTL	Register-Transfer-Logik
SSA	Single-Static-Assignment
SW	Software
XPP	Extrem Processing Platform

Anhang A

Aktivierungsbedingungen für Token-Register

Die hier aufgeführten Bedingungen stellen eine Erweiterung der Beschreibung der Aktivierungsbedingungen für Up- und Down-Register (Abschnitt 4.4.1) dar. Sie werden zur Steuerung der Register verwendet, welche in der HW-Realisierung die Up- und Down-Register des Kontrollers darstellen. Diese Register bleiben solange high wie auch die jeweiligen Aktivierungsbedingungen auf wahr evaluiert sind. Die Namen der einzelnen Bedingungen sind in der Implementierung von COMRADE verwendet worden und können so gut mit dem Source-Code von COMRADE in Verbindung gebracht werden. Die in den Namen enthaltenen Platzhalter haben folgende Bedeutung:

- {FSM}: Die mit diesem Platzhalter beginnende Bedingung wird für jedes Token-Register unabhängig von allen anderen zu einen HW-Modul gehörenden Token-Registern erzeugt.
- {DFG}: Die mit diesem Platzhalter beginnende Bedingung wird für jedes HW-Modul erzeugt und ist für alle Token-Register (die Tokengruppe) eines HW-Moduls dieselbe.

Alle in den folgenden Abbildungen fett umrandeten Bedingungsnahmen werden in eigenständigen Abbildungen genauer spezifiziert.

A.1 Aktivierungsbedingungen der Up-Register

Ein Up-Register ist dann aktiviert, wenn es betreten wird oder weiterhin aktiviert bleiben muss ({FSM}_STATE_UP_ENTER, {FSM}_STATE_UP_HOLD in Bild A.1). Ein Up-Register kann nur betreten werden, wenn weder das Up-Register noch das dazugehörige Down-Register aktiviert ist ({FSM}_ME_ACTIV). Das verhindert, dass ein Up-Register aktiviert wird, wenn das dazugehörige Down-Register aktiv ist. Das Up-Token verbindet sich durch diese Bedingung mit dem Down-Token und beide löschen sich gegenseitig aus. Außer dieser Bedingung muss außerdem die Bedingung {DFG}_COULD_ACTIVATE_UP auftreten.

Das Up-Register wird wiederum solange auch im nächsten Taktzyklus aktiviert, wie es aktiv ist ({FSM}_IN_UP_STATE) und außerdem nicht verlassen werden darf ({FSM}_LEAVE_UP_STATE).

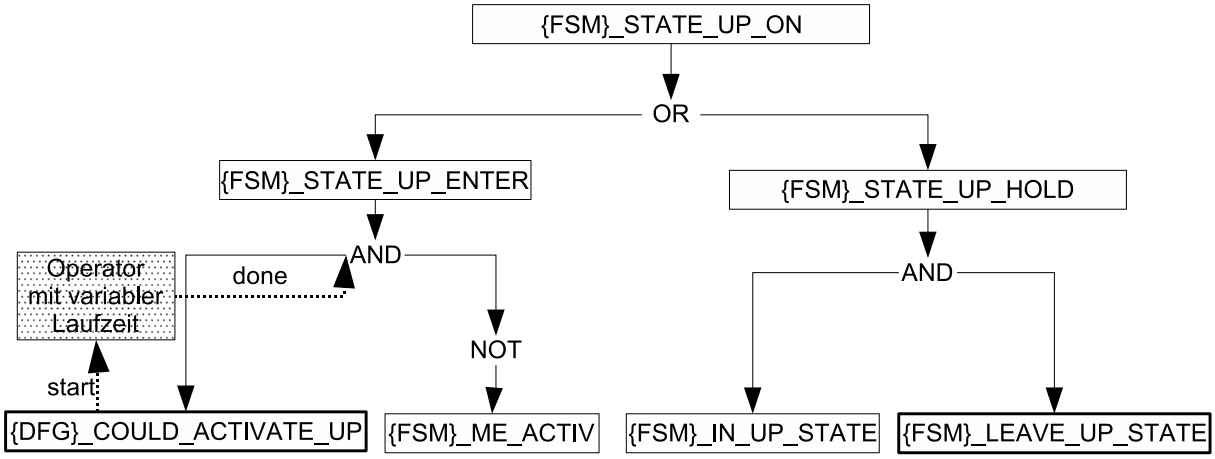


Bild A.1 Aktivierungsbedingung {FSM}_STATE_UP_ON

Ist das Up-Register einem HW-Module mit variabler Laufzeit zugeordnet, dann müssen außerdem noch die start- und done-Signale mit in den Bedingungsbaum aufgenommen werden. Das start-Signal wird hierbei dann mit der Bedingung {DFG}_COULD_ACTIVATE_UP verbunden und startet somit die Ausführung des HW-Moduls. Das Up-Token wird hiermit sozusagen in das HW-Modul übergeben. Das done-Signal wird dann statt {DFG}_COULD_ACTIVATE_UP zur Evaluation der Bedingung verwendet.

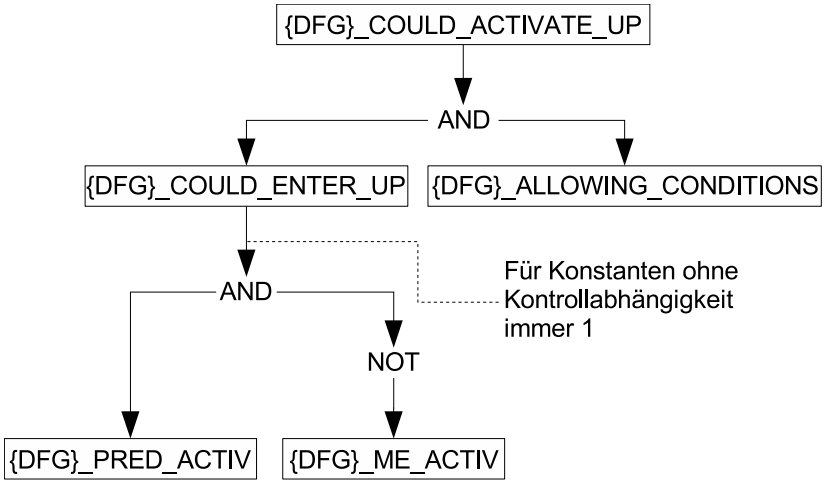


Bild A.2 Aktivierungsbedingung {FSM}_COULD_ACTIVATE_UP

Die Bedingung {DFG}_COULD_ACTIVATE_UP (Bild A.2) wird dann wahr, wenn das Up-Register auf Grund der Up-Register von Vorgänger-HW-Modulen betreten werden kann ({DFG}_COULD_ENTER_UP) und alle Kontroll- und Speicherabhängigkeiten erfüllt sind ({DFG}_ALLOWING_CONDITIONS). Für die Bedingung {DFG}_COULD_ENTER_UP müssen alle Up-Register der Vorgänger-HW-Module aktiv sein, welche die Up-Token für das aktuelle HW-Modul bereitstellen ({DFG}_PRED_ACTIV). Des Weiteren darf keines der Up-Register aktiv sein, welche dem aktuellen HW-Modul zugeordnet sind ({DFG}_ME_ACTIV).

Damit ein HW-Modul aktiviert werden darf, müssen natürlich auch alle Kontroll- und Speicherabhängigkeiten erfüllt worden sein. Diese werden durch die Bedingung {DFG}_ALLOWING_CONDITIONS zusammengefasst. Sie wird wahr, wenn eine der Kontrollabhängigkeiten und eine der Speicherabhängigkeiten erfüllt ist. Eine Kontrollabhängigkeit wird dann erfüllt, wenn das Up-Token eines Vorgänger-HW-Moduls mit Kontrollabhängigkeit bereitsteht und das HW-Modul den mit der Kontrollabhängigkeit verbundenen Wert liefert. Eine Speicherabhängigkeit wird dann erfüllt, wenn das das aktuelle HW-Modul betreffende Up-Register des Vorgänger-HW-Moduls mit Speicherabhängigkeit aktiviert ist.

Ein aktiviertes Up-Register darf deaktiviert werden (Bild A.3), wenn alle Up-Register von nachfolgenden HW-Modulen aktiviert werden dürfen ({FSM}_COULD_ACTIVATE_SUCC_UPS) oder das dazugehörige Down-Register durch eine Bewegung von Down-Token aktiviert werden würde ({FSM}_MOVE_DOWN_ENTER, Abschnitt A.2). Das Up-Register bleibt also bei Ausbleiben beider Bedingungen aktiviert.

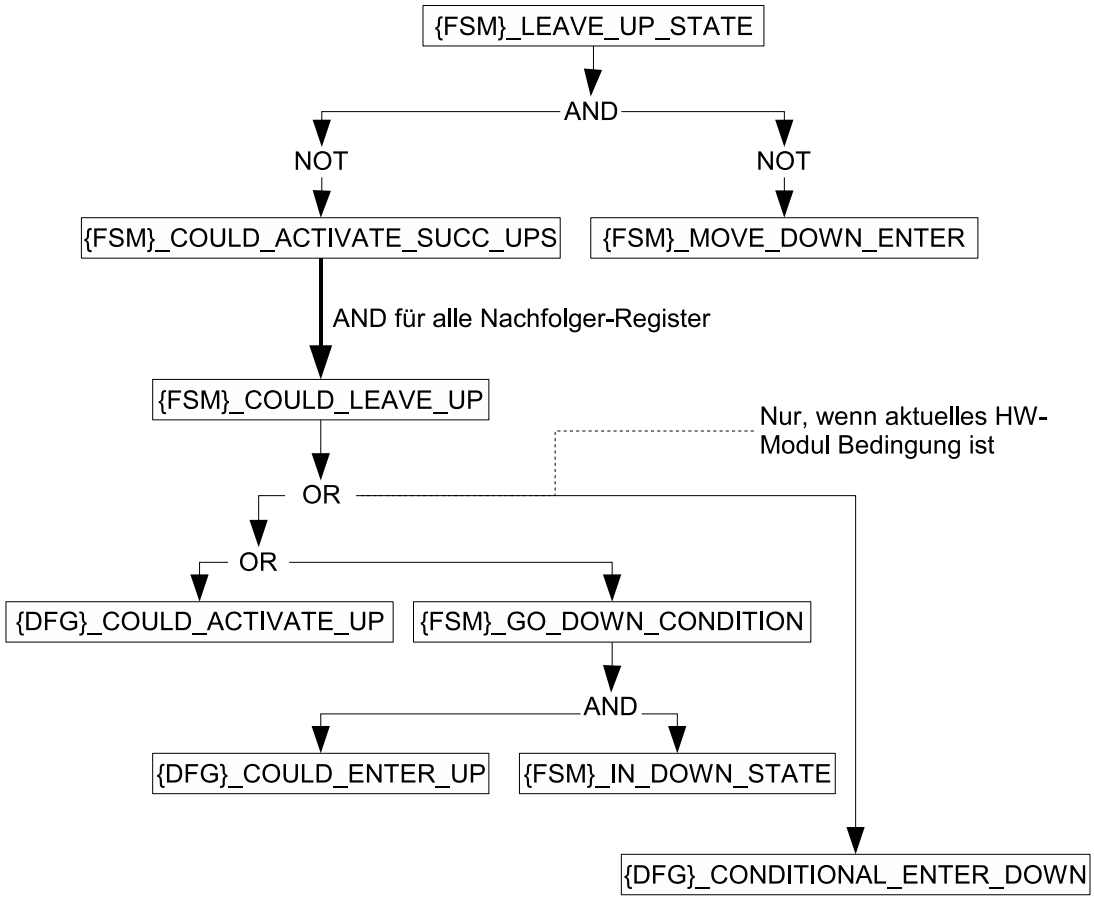


Bild A.3 Aktivierungsbedingung {FSM}_LEAVE_UP_STATE

Ein Nachfolger-Up-Register gilt dann als betretbar, wenn die Bedingung {DFG}_COULD_ACTIVATE_UP für dieses zutrifft oder die Bedingung {FSM}_GO_DOWN_CONDITION wahr ist. Dabei kann man in dieses Up-Register wechseln, wenn auch die Up-Register der kontrollierenden HW-Module aktiv werden könnten

({DFG}_COULD_ENTER_UP) und das zum Up-Register gehörige Down-Register aktiv ist. Im letzteren Fall wird das nachfolgende Up-Register natürlich nicht aktiviert.

Wenn das nachfolgende Register zu einem HW-Modul gehört, welches eine Kontrollinformation erzeugt, so kann es auch aktiviert werden, wenn das Down-Register aktivierbar ist. Dadurch wird das Down-Token auch in Hierarchien von Kontrollinformationen weitergegeben.

A.2 Aktivierungsbedingungen der Down-Register

Die Aktivierungs- und Haltebedingung für ein Down-Register unterscheidet sich von denen der Up-Register dadurch, dass Down-Token auf Grund von Kontrollinformationen gebildet werden und sich entgegen der Up-Token-Richtung bewegen können.

Deshalb ist ein Down-Register aktiv ({FSM}_STATE_DOWN_ON in Bild A.4), wenn es wie auch ein Up-Register betreten ({FSM}_STATE_DOWN_ENTER) oder gehalten wird ({FSM}_STATE_DOWN_HOLD). Entgegen einem Up-Register kann ein Down-Register aber durch das Auftreten einer Kontrollinformation ({FSM}_CONDITIONAL_ENTER_DOWN) oder einer Bewegung ({FSM}_MOVE_DOWN_ACTIVATE) aktiviert werden.

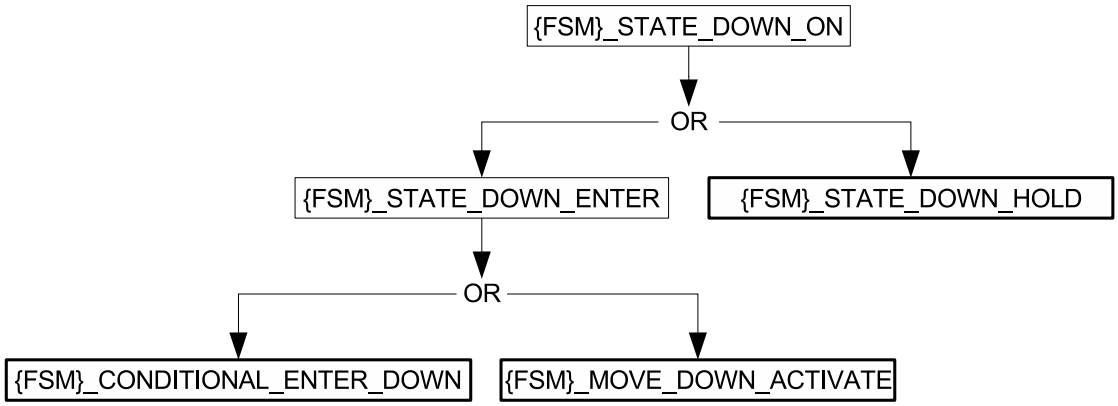


Bild A.4 Aktivierungsbedingung {FSM}_STATE_DOWN_ON

Ein Down-Token wird durch das kontrollabhängige Aktivieren eines Down-Registers erzeugt ({FSM}_CONDITIONAL_ENTER_DOWN in Bild A.5). Dazu dürfen natürlich weder das Up-Register noch das Down-Register aktiviert sein ({DFG}_ME_ACTIV). Außerdem wird das Down-Register nicht aktiviert, wenn es gleichzeitig auch Up-aktiviert werden könnte ({DFG}_COULD_ACTIVATE_UP).

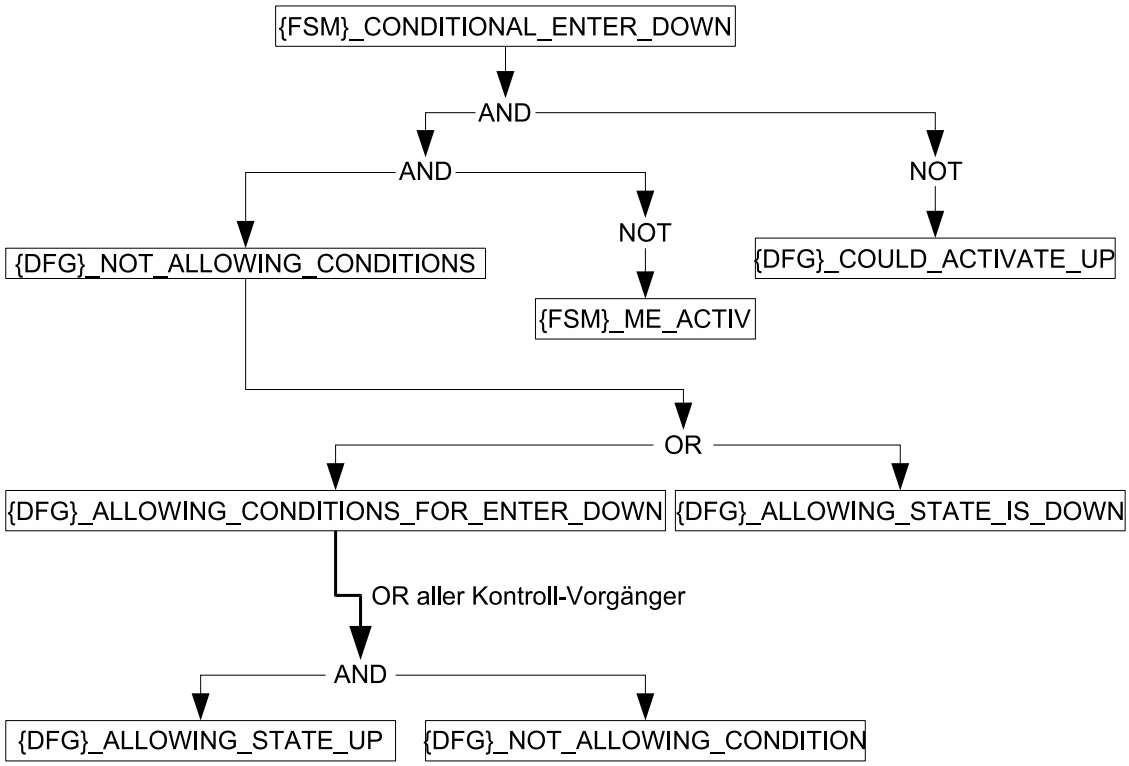


Bild A.5 Aktivierungsbedingung {FSM}_CONDITIONAL_ENTER_DOWN

So löschen sich auch hier bei einem Zusammentreffen Up- und Down-Token aus. Außer diesen beiden Bedingungen müssen natürlich auch die Kontrollinformation für die Aktivierung des Down-Registers vorhanden sein ({DFG}_NOT_ALLOWING_CONDITIONS).

Diese tritt erstens auf, wenn das aktuelle Down-Register zu einem HW-Modul gehört, welches Kontrollinformationen erzeugt, und ein Down-Register eines kontrollierenden HW-Moduls auch aktiv ist ({DFG}_ALLOWING_STATE_IS_DOWN). Außerdem wird ein Down-Register kontrollbedingt aktiviert, wenn ein Up-Register eines kontrollierenden HW-Moduls aktiv ist ({DFG}_ALLOWING_STATE_UP), aber der von diesem HW-Modul generierte Wert nicht das Up-Register aktiviert ({DFG}_ALLOWING_CONDITION).

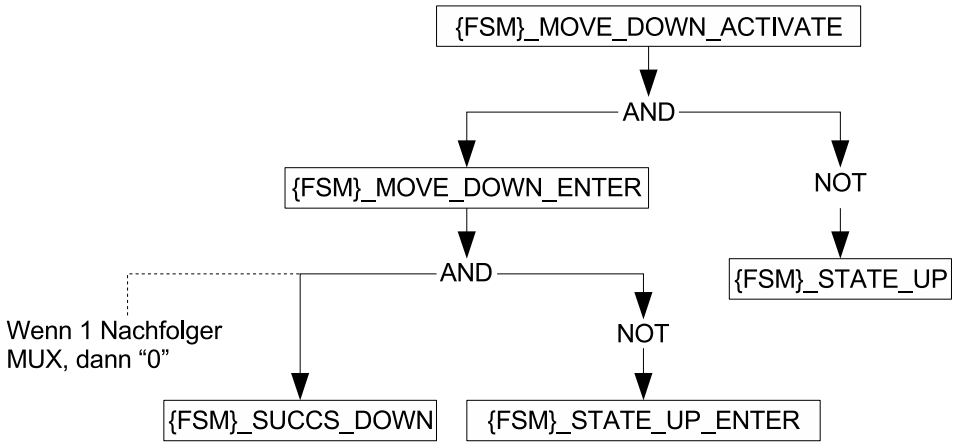


Bild A.6 Aktivierungsbedingung {FSM}_MOVE_DOWN_ACTIVATE

Die zweite Art der Aktivierung eines Down-Registers ist durch eine Bewegung der Down-Tokens ({FSM}_MOVE_DOWN_ACTIVATE in Bild A.6). Ein Down-Register kann nur aktiviert werden, wenn das Up-Register nicht aktiviert ist ({FSM}_STATE_UP). Außerdem müssen alle Nachfolger-Down-Register aktiviert sein ({FSM}_SUCCS_DOWN) und das zum Down-Register gehörende Up-Register darf im nächsten Takt nicht betreten werden ({FSM}_STATE_UP_ENTER).

Ein Down-Register bleibt nach dem Aktivieren ({FSM}_IN_DOWN_STATE) solange aktiv (Bild A.7), bis nicht das dazugehörige Up-Register aktiviert werden ({DFG}_COULD_ENTER_UP) und das Down-Token nicht zu den Vorgängern wandern kann ({FSM}_DOWN_MOVE_TO_PREDS). Das Down-Token bewegt sich dann, wenn alle Vorgängerregister Down-aktiviert werden können. Existiert außerdem noch ein HW-Modul, welches von dem HW-Modul des Down-Registers kontrollabhängig ist, dann muss zusätzlich auch dieses durch eine Kontrollinformation Down-aktiviert werden ({DFG}_CONDITIONAL_ENTER_DOWN).

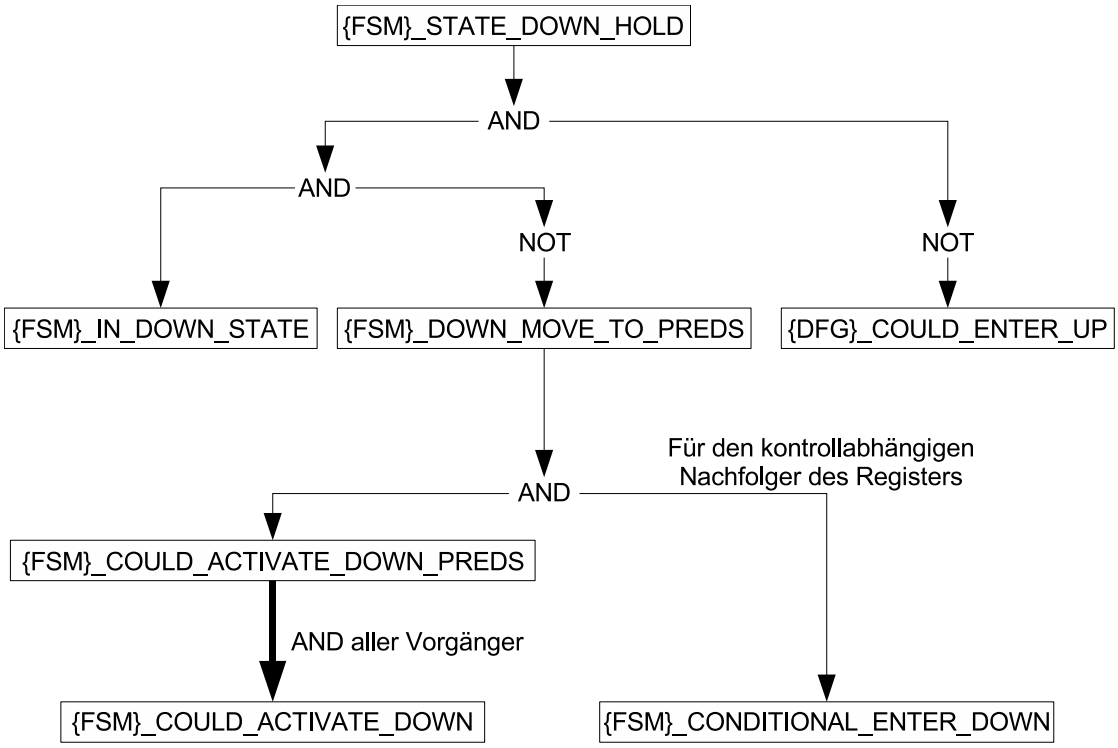


Bild A.7 Aktivierungsbedingung {FSM}_STATE_DOWN_HOLD

Lebenslauf

Name:	Nico Kasprzyk
Geboren am:	12. Juni 1972 in Aschersleben
Familienstand:	verheiratet

Schulausbildung

1979-1989	Zehnklassige Polytechnische Oberschule Ermsleben
1989-1990	Erweiterte Oberschule Aschersleben
1990	Ratsgymnasium Goslar
1990-1993	Gymnasium „Große Schule“ Wolfenbüttel
06.05.1993	Abitur

Wehrdienst

1993-1994	Wehrdienst

Studium

1994-1999	Studium der Informatik an der Technischen Universität Braunschweig
29.09.1999	Diplom der Informatik

Berufstätigkeit

1997	Studentische Hilfskraft am Institut für Wissenschaftliches Rechnen der Technischen Universität Braunschweig
1997-1999	Studentische Hilfskraft an der Abteilung Entwurf integrierter Schaltungen (E.I.S.) der Technischen Universität Braunschweig
1998	Praktikum am Institut für Mathematische Maschinen, Kiew
1999	Unabhängiger Software-Entwickler
1999-2004	Wissenschaftlicher Mitarbeiter an der Abteilung Entwurf integrierter Schaltung der Technischen Universität Braunschweig
2000	Praktikum in der Advanced Technologie Group bei Synopsys (Mountain View, Kalifornien)